Learn the Java skills you will need to start developing Android apps

Learn Java for Android Development

THIRD EDITION

Jeff Friesen

Apress®

www.it-ebooks.info
For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.
Contents at a Glance

About the Author ................................................................................................................................. xxii
About the Technical Reviewer ........................................................................................................... xxiii
Acknowledgments ............................................................................................................................... xxv
Introduction ........................................................................................................................................... xxvii

Chapter 1: Getting Started with Java ................................................................................................. 1
Chapter 2: Learning Language Fundamentals .................................................................................. 31
Chapter 3: Discovering Classes and Objects .................................................................................... 89
Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces .................................................. 141
Chapter 5: Mastering Advanced Language Features, Part 1 .............................................................. 189
Chapter 6: Mastering Advanced Language Features, Part 2 ............................................................... 235
Chapter 7: Exploring the Basic APIs, Part 1 ...................................................................................... 287
Chapter 8: Exploring the Basic APIs, Part 2 ...................................................................................... 359
Chapter 9: Exploring the Collections Framework .............................................................................. 401
Chapter 10: Exploring the Concurrency Utilities .............................................................................. 487
Chapter 11: Performing Classic I/O .................................................................................................... 539
Chapter 12: Accessing Networks ....................................................................................................... 621
Introduction

Smartphones and tablets are all the rage these days. Their popularity is largely due to their ability to run apps. Although the iPhone and iPad, with their growing collection of Objective-C based apps, had a head start, Android-based smartphones and tablets, with their growing collection of Java-based apps, have proven to be a strong competitor.

Not only are many iPhone/iPad app developers making money by selling their apps, but many Android app developers are also making money by selling similar apps. According to tech web sites such as The Register (www.theregister.co.uk), some Android app developers are making lots of money (www.theregister.co.uk/2010/03/02/android_app_profit).

In today’s challenging economic climate, you might like to try your hand at developing Android apps and make some money. If you have good ideas, perseverance, and some artistic talent (or perhaps know some talented individuals), you are already part of the way toward achieving this goal.

Tip A good reason to consider Android app development over iPhone/iPad app development is the lower startup costs that you’ll incur with Android. For example, you don’t need to purchase a Mac on which to develop Android apps (a Mac is required for developing iPhone/iPad apps); your existing Windows, Linux, or Unix machine will do nicely.

Most importantly, you’ll need to possess a solid understanding of the Java language and foundational application programming interfaces (APIs) before jumping into Android. After all, Android apps are written in Java and interact with many of the standard Java APIs (such as threading and input/output APIs).

I wrote Learn Java for Android Development to give you a solid Java foundation that you can later extend with knowledge of Android architecture, API, and tool specifics. This book will give you a strong grasp of the Java language and the many important APIs that are fundamental to Android apps and other Java applications. It will also introduce you to key development tools.
Book Organization

The first edition of this book was organized into 10 chapters and 1 appendix. The second edition was organized into 14 chapters and 3 appendixes. This third edition is organized into 16 chapters and 2 appendixes with a bonus appendix on Android app development. Each chapter in each edition offers a set of exercises that you should complete to get the most benefit from its content. Their solutions are presented in Appendix A.

Chapter 1 introduces you to Java by first focusing on Java’s dual nature (language and platform). It then briefly introduces you to Oracle’s Java SE, Java EE, and Java ME editions of the Java platform. You next learn how to download and install the Java SE Development Kit (JDK), and you learn some Java basics by developing and playing with three simple Java applications. After receiving a brief introduction to the Eclipse IDE, you receive a brief introduction to Android.

Chapter 2 starts you on an in-depth journey of the Java language by focusing on language fundamentals. You first learn about simple application structure and then learn about comments, identifiers (and reserved words), types, variables, expressions (and literals), and statements.

Chapter 3 continues your journey by focusing on classes and objects. You learn how to declare a class and organize applications around multiple classes. You then learn how to construct objects from classes, declare fields in classes and access these fields, declare methods in classes and call them, initialize classes and objects, and remove objects when they’re no longer needed. You also learn more about arrays, which were first introduced in Chapter 2.

Chapter 4 adds to Chapter 3’s pool of object-based knowledge by introducing you to the language features that take you from object-based applications to object-oriented applications. Specifically, you learn about features related to inheritance, polymorphism, and interfaces. While exploring inheritance, you learn about Java’s ultimate superclass. Also, while exploring interfaces, you discover why they were included in the Java language; interfaces are not merely a workaround for Java’s lack of support for multiple implementation inheritance, but serve a higher purpose.

Chapter 5 introduces you to four categories of advanced language features: nested types, packages, static imports, and exceptions.

Chapter 6 introduces you to four additional advanced language feature categories: assertions, annotations, generics, and enums.

Chapter 7 begins a trend that focuses more on APIs than language features. This chapter first introduces you to Java’s Math and StrictMath math-oriented types. It then explores Number and its various subtypes (such as Integer, Double, and BigDecimal). Next you explore the string-oriented types (String, StringBuffer, and StringBuilder) followed by the System type. Finally, you explore the Thread class and related types for creating multithreaded applications.

Chapter 8 continues to explore Java’s basic APIs by focusing on the Random class for generating random numbers; the References API, Reflection, the StringTokenizer class for breaking a string into smaller components; and the Timer and TimerTask classes for occasionally or repeatedly executing tasks.

Chapter 9 focuses exclusively on Java’s Collections Framework, which provides you with a solution for organizing objects in lists, sets, queues, and maps. You also learn about collection-oriented utility classes and review Java’s legacy collection types.
Chapter 10 focuses exclusively on Java’s Concurrency Utilities. After receiving an introduction to this framework, you explore executors, synchronizers (such as countdown latches), concurrent collections, the Locking Framework, and atomic variables (where you discover compare-and-swap).

Chapter 11 is all about classic input/output (I/O), largely from a file perspective. In this chapter, you explore classic I/O in terms of the File class, RandomAccessFile class, various stream classes, and various writer/reader classes. My discussion of stream I/O includes coverage of Java’s object serialization and deserialization mechanisms.

Chapter 12 continues to explore classic I/O by focusing on networks. You learn about the Socket, ServerSocket, DatagramSocket, and MulticastSocket classes along with related types. You also learn about the URL class for achieving networked I/O at a higher level and learn about the related URI class. After learning about the low-level NetworkInterface and InterfaceAddress classes, you explore cookie management, in terms of the CookieHandler and CookieManager classes, and the CookiePolicy and CookieStore interfaces.

Chapter 13 introduces you to New I/O. You learn about buffers, channels, selectors, regular expressions, charsets, and the Formatter and Scanner types in this chapter.

Chapter 14 focuses on databases. You first learn about the Java DB and SQLite database products, and then explore JDBC for communicating with databases created via these products.

Chapter 15 emphasizes Java’s support for XML. I first provide a tutorial on this topic where you learn about the XML declaration, elements and attributes, character references and CDATA sections, namespaces, comments and processing instructions, well-formed documents, and valid documents (in terms of Document Type Definition and XML Schema). I then show you how to parse XML documents via the SAX API, parse and create XML documents via the DOM API, parse XML documents via the XMLPULL V1 API (supported by Android as an alternative to Java’s StAX API), use the XPath API to concisely select nodes via location path expressions, and transform XML documents via XSLT.

Chapter 16 completes the chapter portion of this book by covering odds and ends. You first learn about useful Java 7 language features that I’ve successfully used in Android apps. Next, you explore classloaders, the Console class, design patterns (with emphasis on the Strategy pattern), double brace initialization, fluent interfaces, immutability, internationalization (in terms of locales; resource bundles; break iterators; collators; dates, time zones, and calendars; and formatters), the Logging API, the Preferences API, the Runtime and Process classes, the Java Native Interface, and the ZIP and JAR APIs.

Appendix A presents solutions to all of the exercises in Chapters 1 through 16.

Appendix B introduces you to application development in the context of Four of a Kind, a console-based card game.

Appendix C provides an introduction to Android app development. It gives you a chance to see how various Java language features and APIs are used in an Android context.

Unlike the other elements, Appendix C is not included in this book—it’s included with the book’s source code. Appendix C doesn’t officially belong in Learn Java for Android Development because this book’s focus is to prepare you for getting into Android app development by teaching you the fundamentals of the Java language, and Appendix C goes beyond that focus by giving you a tutorial on Android app development. Besides, the presence of this appendix would cause the book to exceed the 1,200-page print-on-demand limit.
Note  You can download this book’s source code by pointing your web browser to www.apress.com/9781430264545 and clicking the Source Code tab followed by the Download Now link.

What Comes Next?

After you complete this book, I recommend that you check out Apress’s other Android-oriented books, such as Beginning Android 4 by Grant Allen (Apress, 2012), and learn more about developing Android apps. In that book, you learn Android basics and how to create “innovative and salable applications for Android 4 mobile devices.”

Thanks for purchasing this third (and my final) edition of Learn Java for Android Development. I hope you find it a helpful preparation for, and I wish you lots of success in achieving, a satisfying and lucrative career as an Android app developer.

—Jeff Friesen, January 2014
Getting Started with Java

Android apps are written in Java and use various Java application program interfaces (APIs). Because you'll want to write your own apps, but may be unfamiliar with the Java language and these APIs, this book teaches you about Java as a first step into Android app development. It provides you with Java language fundamentals and Java APIs that are useful when developing apps.

Note This book illustrates Java concepts via non-Android Java applications. It's easier for beginners to grasp these applications than corresponding Android apps. However, I also reveal a trivial Android app toward the end of this chapter for comparison purposes.

An API is an interface that application code uses to communicate with other code, which is typically stored in a software library. For more information on this term, check out Wikipedia’s “Application programming interface” topic at http://en.wikipedia.org/wiki/Application_programming_interface.

This chapter sets the stage for teaching you the essential Java concepts that you need to understand before embarking on an Android app development career. I first answer the question: “What is Java?” Next, I show you how to install the Java SE Development Kit (JDK) and introduce you to JDK tools for compiling and running Java applications.

After presenting a few simple example applications, I show you how to install and use the open source Eclipse IDE (integrated development environment) so that you can more easily (and more quickly) develop Java applications and (eventually) Android apps. I then provide you with a brief introduction to Android and show you how Java fits into the Android development paradigm.
What Is Java?

Java is a language and a platform originated by Sun Microsystems. In this section, I briefly describe this language and reveal what it means for Java to be a platform. To meet various needs, Sun organized Java into three main editions: Java SE, Java EE, and Java ME. This section briefly explores each of these editions.

Note  Java has an interesting history that dates back to December 1990. At that time, James Gosling, Patrick Naughton, and Mike Sheridan (all employees of Sun Microsystems) were given the task of figuring out the next major trend in computing. They concluded that one trend would involve the convergence of computing devices and intelligent consumer appliances. Thus was born the Green Project.

The fruits of Green were Star7, a handheld wireless device featuring a five-inch color LCD screen, a SPARC processor, a sophisticated graphics capability, a version of Unix, and Oak, a language developed by James Gosling for writing applications to run on Star7 that he named after an oak tree growing outside of his office window at Sun. To avoid a conflict with another language of the same name, Dr. Gosling changed this language’s name to Java.


Java Is a Language

Java is a language in which developers express source code (program text). Java’s syntax (rules for combining symbols into language features) is partly patterned after the C and C++ languages in order to shorten the learning curve for C/C++ developers.

The following list identifies a few similarities between Java and C/C++:

- Java and C/C++ share the same single-line and multi-line comment styles. Comments let you document source code.
- Many of Java’s reserved words are identical to their C/C++ counterparts (for, if, switch, and while are examples) and C++ counterparts (catch, class, public, and try are examples).
- Java supports character, double precision floating-point, floating-point, integer, long integer, and short integer primitive types via the same char, double, float, int, long, and short reserved words.
- Java supports many of the same operators, including arithmetic (+, -, *, /, and %) and conditional (?:) operators.
- Java uses brace characters ({ and }) to delimit blocks of statements.
The following list identifies a few of the differences between Java and C/C++:

- Java supports an additional comment style known as Javadoc.
- Java provides reserved words not found in C/C++ (extends, strictfp, synchronized, and transient are examples).
- Java doesn’t require machine-specific knowledge. It supports the byte integer type (see http://en.wikipedia.org/wiki/Integer_(computer_science)), doesn’t provide a signed version of the character type, and doesn’t provide unsigned versions of integer, long integer, and short integer. Furthermore, all of Java’s primitive types have guaranteed implementation sizes, which is an important part of achieving portability (discussed later). The same cannot be said of equivalent primitive types in C and C++.
- Java provides operators not found in C/C++. These operators include instanceof and >>> (unsigned right shift).
- Java provides labeled break and continue statements that you’ll not find in C/C++.

You’ll learn about single-line, multi-line, and Javadoc comments in Chapter 2. Also, you’ll learn about reserved words, primitive types, operators, blocks, and statements (including labeled break and labeled continue) in that chapter.

Java was designed to be a safer language than C/C++. It achieves safety in part by not letting you overload operators and by omitting C/C++ features such as pointers (storage locations containing addresses; see http://en.wikipedia.org/wiki/Pointer_(computer_programming)).

Java also achieves safety by modifying certain C/C++ features. For example, loops must be controlled by Boolean expressions instead of integer expressions where 0 is false and a nonzero value is true. (There is a discussion of loops and expressions in Chapter 2.)

Suppose you must code a C/C++ while loop that repeats no more than 10 times. Being tired, you specify the following:

```java
while (x)
    x++;
```

Assume that x is an integer-based variable initialized to 0 (I discuss variables in Chapter 2). This loop repeatedly executes x++ to add 1 to x’s value. This loop doesn’t stop when x reaches 10; you have introduced a bug.

This problem is less likely to occur in Java because it complains when it sees while (x). This complaint requires you to recheck your expression, and you will then most likely specify while (x != 10). Not only is safety improved (you cannot specify just x), but meaning is also clarified: while (x != 10) is more meaningful than while (x).

These and other fundamental language features support classes, objects, inheritance, polymorphism, and interfaces. Java also provides advanced features related to nested types, packages, static imports, exceptions, assertions, annotations, generics, enums, and more. Subsequent chapters explore most of these language features.
Java Is a Platform

Java is a platform consisting of a virtual machine and an execution environment. The virtual machine is a software-based processor that presents an instruction set, and it is commonly referred to as the Java Virtual Machine (JVM). The execution environment consists of libraries for running programs and interacting with the underlying operating system (also known as the native platform).

The execution environment includes a huge library of prebuilt classfiles that perform common tasks, such as math operations (trigonometry, for example) and network communications. This library is commonly referred to as the standard class library.

A special Java program known as the Java compiler translates source code into object code consisting of instructions that are executed by the JVM and associated data. These instructions are known as bytecode. Figure 1-1 shows this translation process.

![Figure 1-1. The Java compiler translates Java source code into Java object code consisting of bytecode and associated data](image)

The compiler stores a program’s bytecode and data in files having the .class extension. These files are known as classfiles because they typically store the compiled equivalent of classes, a language feature discussed in Chapter 3. Figure 1-2 shows the organization of a classfile.

![Figure 1-2. A classfile is organized into a magic number, version number, constant pool, and seven other sections](image)

Don’t worry about having to know this classfile architecture. I present it to satisfy the curiosities of those who are interested in learning more about how classfiles are organized.

A Java program executes via a tool that loads and starts the JVM and passes the program’s main classfile to the machine. The JVM uses its classloader component to load the classfile into memory.
After the classfile has been loaded, the JVM’s *bytecode verifier* component makes sure that the classfile’s bytecode is valid and doesn’t compromise security. The verifier terminates the JVM when it finds a problem with the bytecode.

Assuming that all is well with the classfile’s bytecode, the JVM’s *interpreter* component interprets the bytecode one instruction at a time. *Interpretation* consists of identifying bytecode instructions and executing equivalent native instructions.

Note  Native instructions (also known as native code) are the instructions understood by the native platform’s physical processor.

When the interpreter learns that a sequence of bytecode instructions is executed repeatedly, it informs the JVM’s *just-in-time (JIT) compiler* to compile these instructions into native code.

JIT compilation is performed only once for a given sequence of bytecode instructions. Because the native instructions execute instead of the associated bytecode instruction sequence, the program executes much faster.

During execution, the interpreter might encounter a request to execute another classfile’s bytecode. When that happens, it asks the classloader to load the classfile and the bytecode verifier to verify the bytecode before executing that bytecode.

Also during execution, bytecode instructions might request that the JVM open a file, display something on the screen, or perform another task that requires cooperation with the native platform. The JVM responds by transferring the request to the platform via its *Java Native Interface (JNI)* bridge to the native platform. Figure 1-3 shows these JVM tasks.
The platform side of Java promotes *portability* by providing an abstraction over the underlying platform. As a result, the same bytecode runs unchanged on Windows, Linux, Mac OS X, and other platforms.

**Note** Java was introduced with the slogan “write once, run anywhere.” Although Java goes to great lengths to enforce portability (such as defining an integer always to be 32 binary digits [bits] and a long integer always to be 64 bits (see http://en.wikipedia.org/wiki/Bit to learn about binary digits), it doesn’t always succeed. For example, despite being mostly platform independent, certain parts of Java (such as the scheduling of threads, discussed in Chapter 7) vary from underlying platform to underlying platform.

The platform side of Java also promotes *security* by doing its best to provide a secure environment (such as the bytecode verifier) in which code executes. The goal is to prevent malicious code from corrupting the underlying platform (and possibly stealing sensitive information).

**Note** Many security issues that have plagued Java have prompted Oracle to release various security updates. For example, blogger Brian Krebs reported on a recent update (at time of this writing) that fixes 51 security issues in his “Critical Java Update Plugs 51 Security Holes” blog post (http://krebsonsecurity.com/2013/10/java-update-plugs-51-security-holes/). Although troubling, Oracle is keeping on top of this ongoing problem (whose impact on Android is minimal).

### Java SE, Java EE, and Java ME

Developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices (such as feature phones), and embedded devices (such as television set-top boxes).

- **Java Platform, Standard Edition (Java SE):** The Java platform for developing *applications*, which are stand-alone programs that run on desktops. Java SE is also used to develop *applets*, which are programs that run in web browsers.

- **Java Platform, Enterprise Edition (Java EE):** The Java platform for developing enterprise-oriented applications and *servlets*, which are server programs that conform to Java EE’s Servlet API. Java EE is built on top of Java SE.

- **Java Platform, Micro Edition (Java ME):** The Java platform for developing *MIDlets*, which are programs that run on mobile information devices, and *Xlets*, which are programs that run on embedded devices.

This book largely focuses on Java SE and applications.
CHAPTER 1: Getting Started with Java

Installing the JDK and Exploring Example Applications

The Java Runtime Environment (JRE) implements the Java SE platform and makes it possible to run Java programs. The public JRE can be downloaded from Oracle’s Java SE Downloads page at www.oracle.com/technetwork/java/javase/downloads/index.html.

However, the public JRE doesn’t make it possible to develop Java (and Android) applications. You need to download and install the Java SE Development Kit (JDK), which contains development tools (including the Java compiler) and a private JRE.

Note: Oracle is also championing Java Embedded, a collection of technologies that brings Java to all kinds of devices (such as smartcards and vehicle navigation systems). Java SE Embedded and Java ME Embedded are the two major subsets of Java Embedded.

Note: JDK 1.0 was the first JDK to be released (in May 1995). Until JDK 6 arrived, JDK stood for Java Development Kit (SE wasn’t part of the title). Over the years, numerous JDKs have been released, with JDK 7 being current at time of this writing.

Each JDK’s version number identifies a version of Java. For example, JDK 1.0 identifies Java 1.0, and JDK 5 identifies Java 5.0. JDK 5 was the first JDK also to provide an internal version number: 1.5.0.

The Java SE Downloads page also provides access to the current JDK, which is JDK 7 Update 45 at time of this writing. Click the appropriate Download button to download the current JDK’s installer application for your platform. Then run this application to install the JDK.

The JDK installer places the JDK in a home directory. (It can also install the public JRE in another directory.) On my Windows 7 platform, the home directory is C:\Program Files\Java\jdk1.7.0_06. (I currently use JDK 7 Update 6—I’m slow to upgrade.)

Tip: After installing the JDK, you should add the bin subdirectory to your platform’s PATH environment variable (see http://java.com/en/download/help/path.xml) so that you can execute JDK tools from any directory. Also, you might want to create a projects subdirectory of the JDK’s home directory to organize your Java projects and create a separate subdirectory within projects for each of these projects.
The home directory contains various files (such as README.html, which provides information about the JDK, and src.zip, which provides the standard class library source code) and subdirectories, including the following three important subdirectories:

- **bin**: This subdirectory contains assorted JDK tools. You’ll use only a few of these tools in this book, mainly javac (Java compiler) and java (Java application launcher). However, you’ll also work with jar (Java ARchive [JAR] creator, updater, and extractor—a JAR file is a ZIP file with special features), javadoc (Java documentation generator), and serialver (serial version inspector).

- **jre**: This subdirectory contains the JDK’s private copy of the JRE, which lets you run Java programs without having to download and install the public JRE.

- **lib**: This subdirectory contains library files that are used by JDK tools. For example, tools.jar contains the Java compiler’s classfiles. The compiler was written in Java.

**Note** javac isn’t the Java compiler. It’s a tool that loads and starts the JVM, identifies the compiler’s main classfile (located in tools.jar) to the JVM, and passes the name of the source file being compiled to the compiler’s main classfile.


The following command line shows you how to use javac to compile a source file named App.java:

```
javac App.java
```

You can compile multiple source files by specifying an asterisk in place of the filename, as follows:

```
javac *.java
```

Assuming success, an App.class file is created. If this file describes an application, which minimally consists of a single class containing a method named main, you can run the application as follows:

```
java App
```

You must not specify the .class file extension. The java tool complains when .class is specified.

In addition to downloading and installing the JDK, you’ll need to access the JDK documentation, especially to explore the Java APIs. There are two sets of documentation that you can explore.
Oracle’s JDK 7 documentation (http://docs.oracle.com/javase/7/docs/api/index.html)


Oracle’s JDK 7 documentation presents many APIs that are not supported by Android. Furthermore, it doesn’t cover APIs that are specific to Android. This book focuses only on core Oracle Java APIs that are also covered in Google’s documentation.

**Hello, World!**

It’s customary to start exploring a new language and its tools by writing, compiling, and running a simple application that outputs the “Hello, World!” message. This practice dates back to Brian Kernighan’s and Dennis Ritchie’s seminal book, *The C Programming Language*.

Listing 1-1 presents the source code to a HelloWorld application that outputs this message.

Listing 1-1. *Saying Hello in a Java Language Context*

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

This short seven-line application has a lot to say about the Java language. I’ll briefly explain each feature, leaving comprehensive discussions of these features to later chapters.

This source code declares a `class`, which you can think of as a container for describing an application. The first line, `public class HelloWorld`, introduces the name of the class (HelloWorld), which is preceded by `reserved words` (names that have meaning to the Java compiler and which you cannot use to name other things in your programs) `public` and `class`. These reserved words respectively tell the compiler that HelloWorld must be stored in a file named HelloWorld and that a class is being declared.

The rest of the class declaration appears between a pair of brace characters ({}), which are familiar to C and C++ developers. Between these characters is the declaration of a single `method`, which you can think of as a named sequence of code. This method is named `main` to signify that it’s the entry point into the application, and it is the analog of the `main()` function in C and C++.
The `main()` method includes a header that identifies this method and a block of code located between an open brace character (`{`) and a close brace character (`}`). Besides naming this method, the header provides the following information:

- **public**: This reserved word makes `main()` visible to the startup code that calls this method. If `public` wasn’t present, the compiler would output an error message stating that it couldn’t find a `main()` method.

- **static**: This reserved word causes this method to associate with the class instead of associating with any objects (discussed in Chapter 3) created from this class. Because the startup code that calls `main()` doesn’t create an object from the class to call this method, it requires that the method be declared static. Although the compiler will not report an error when `static` is missing, it will not be possible to run `HelloWorld`, which will not be an application when the proper `main()` method doesn’t exist.

- **void**: This reserved word indicates that the method doesn’t return a value. If you change `void` to a type’s reserved word (such as `int`) and then insert code that returns a value of this type (such as `return 0;`), the compiler will not report an error. However, you won’t be able to run `HelloWorld` because the proper `main()` method wouldn’t exist. (I discuss types in Chapter 2.)

- **(String[] args)**: This parameter list consists of a single parameter named `args`, which is of type `String[]`. Startup code passes a sequence of command-line arguments to `args`, which makes these arguments available to the code that executes within `main()`. You’ll learn about parameters and arguments in Chapter 3.

`main()` is called with an array of `strings` (character sequences delimited by double quote " characters) that identify the application’s command-line arguments. These strings are stored in `String`-based array variable `args`. (I discuss method calling, arrays, and variables in Chapters 2 and 3.) Although the array variable is named `args`, there’s nothing special about this name. You could choose another name for this variable.

`main()` presents a single line of code, `System.out.println("Hello, World!");`, which is responsible for outputting `Hello, World!` in the command window from where `HelloWorld` is run. From left to right, this *method call* accomplishes the following tasks:

- **System** identifies a standard class of system utilities.

- **out** identifies an object variable located in `System` whose methods let you output values of various types optionally followed by a newline (also known as line feed) character to the standard output stream. (In reality, a platform-dependent line terminator sequence is output. On Windows platforms, this sequence consists of a carriage return character [integer value 13] followed by a line feed character [integer value 10]. On Linux platforms, this sequence consists of a line feed character. On Mac OS X systems, this sequence consists of a carriage return character. It’s convenient to refer to this sequence as a newline.)

- **println** identifies a method that prints its "Hello, World!" argument (the starting and ending double quote characters are not written; these characters delimit but are not part of the string) followed by a newline to the standard output stream.
Note The standard output stream is part of Standard I/O ([http://en.wikipedia.org/wiki/Standard_streams](http://en.wikipedia.org/wiki/Standard_streams)), which also consists of standard input and standard error streams, and which originated with the Unix operating system. Standard I/O makes it possible to read text from different sources (keyboard or file) and write text to different destinations (screen or file).

Text is read from the standard input stream, which defaults to the keyboard but can be redirected to a file. Text is written to the standard output stream, which defaults to the screen but can be redirected to a file. Error message text is written to the standard error stream, which defaults to the screen but can be redirected to a file that differs from the standard output file.

Assuming that you’re familiar with your platform’s command-line interface and are at the command line, make HelloWorld your current directory and copy Listing 1-1 to a file named HelloWorld.java. Then compile this source file via the following command line:

```
javac HelloWorld.java
```

Assuming that you’ve included the .java extension, which is required by javac, and that HelloWorld.java compiles, you should discover a file named HelloWorld.class in the current directory. Run this application via the following command line:

```
java HelloWorld
```

If all goes well, you should see the following line of output on the screen:

```
Hello, World!
```

You can redirect this output to a file by specifying the greater than angle bracket (>) followed by a filename. For example, the following command line stores the output in a file named hello.txt:

```
java HelloWorld >hello.txt
```

**DumpArgs**

In the previous example, I pointed out main()’s (String[] args) parameter list, which consists of a single parameter named args. This parameter stores an array (think sequence of values) of arguments passed to the application on the command line. Listing 1-2 presents the source code to a DumpArgs application that outputs each argument.

```
Listing 1-2. Dumping Command-Line Arguments Stored in main()’s args Array to the Standard Output Stream
```

class DumpArgs
{
    public static void main(String[] args)
    {
        System.out.println("Passed arguments:");
        for (int i = 0; i < args.length; i++)
        {
Listing 1-2’s DumpArgs application consists of a class named DumpArgs that’s very similar to Listing 1-1’s HelloWorld class. The essential difference between these classes is the for loop (a construct for repeated execution and starting with reserved word for) that accesses each array item and dumps it to the standard output stream.

The for loop first initializes integer variable i to 0. This variable keeps track of how far the loop has progressed (the loop must end at some point), and it also identifies one of the entries in the args array. Next, i is compared with args.length, which records the number of entries in the array. The loop ends when i’s value equals the value of args.length. (I discuss .length in Chapter 2.)

Each loop iteration executes System.out.println(args[i]);. The string stored in the i-th entry of the args array is accessed and then output to the standard output stream—the first entry is located at index (location) 0. The last entry is stored at index args.length - 1. Finally, i is incremented by 1 via i++, and i < args.length is reevaluated to determine whether the loop continues or ends.

Assuming that you’re familiar with your platform’s command-line interface and that you are at the command line, make DumpArgs your current directory and copy Listing 1-2 to a file named DumpArgs.java. Then compile this source file via the following command line:

```
javac DumpArgs.java
```

Assuming that that you’ve included the .java extension, which is required by javac, and that DumpArgs.java compiles, you should discover a file named DumpArgs.class in the current directory. Run this application via the following command line:

```
java DumpArgs
```

If all goes well, you should see the following line of output on the screen:

```
Passed arguments:
```

For more interesting output, you’ll need to pass command-line arguments to DumpArgs. For example, execute the following command line, which specifies Curly, Moe, and Larry as three arguments to pass to DumpArgs:

```
java DumpArgs Curly Moe Larry
```

This time, you should see the following expanded output on the screen:

```
Passed arguments:
Curly
Moe
Larry
```
You can redirect this output to a file. For example, the following command line stores the DumpArgs application’s output in a file named out.txt:

```
java DumpArgs Curly Moe Larry >out.txt
```

**EchoText**

The previous two examples introduced you to a few Java language features, and they also showed outputting text to the standard output stream, which defaults to the screen but can be redirected to a file. In the final example (see Listing 1-3), I introduce more language features and demonstrate inputting text from the standard input stream and outputting text to the standard error stream.

**Listing 1-3. Echoing Text Read from Standard Input to Standard Output**

```
public class EchoText
{
    public static void main(String[] args)
    {
        boolean isRedirect = false;
        if (args.length != 0)
            isRedirect = true;
        int ch;
        try
        {
            while ((ch = System.in.read()) != ((isRedirect) ? -1 : '\n'))
                System.out.print((char) ch);
        }
        catch (java.io.IOException ioe)
        {
            System.err.println("I/O error");
        }
        System.out.println();
    }
}
```

EchoText is a more complex application than HelloWorld or DumpArgs. Its main() method first declares a Boolean (true/false) variable named isRedirect that tells this application whether input originates from the keyboard (isRedirect is false) or a file (isRedirect is true). The application defaults to assuming that input originates from the keyboard.

There’s no easy way to determine if standard input has been redirected, and so the application requires that the user tell it if this is the case by specifying one or more command-line arguments. The if decision (a construct for making decisions and starting with reserved word if) evaluates args.length != 0, assigning true to isRedirect when this Boolean expression evaluates to true (at least one command-line argument has been specified).

main() now introduces the int variable ch to store the integer representation of each character read from standard input. (You'll learn about int and integer in Chapter 2.) It then enters a sequence of code prefixed by the reserved word try and surrounded by brace characters. Code within this block may throw an exception (an object describing a problem) and the subsequent catch block will handle it (to address the problem). (I discuss exceptions in Chapter 5.)
The try block consists of a *while loop* (a construct for repeated execution and starting with the reserved word `while`) that reads and echoes characters. The loop first calls `System.in.read()` to read a character and assign its integer value to `ch`. The loop ends when this value equals -1 (no more input data is available from a file; standard input was redirected) or '
' (the newline/line feed character has been read, which is the case when standard input wasn't redirected.) '
' is an example of a character literal, which is discussed in Chapter 2.

For any other value in `ch`, this value is converted to a character via `(char)`, which is an example of Java's cast operator (discussed in Chapter 2). The character is then output via `System.out.print()`, which doesn’t also terminate the current line by outputting a newline. The final `System.out.println()` call terminates the current line without outputting any content.

When standard input is redirected to a file and `System.in.read()` is unable to read text from the file (perhaps the file is stored on a removable storage device that has been removed before the read operation), `System.in.read()` fails by throwing a `java.io.IOException` object that describes this problem. The code within the catch block is then executed, which outputs an I/O error message to the standard error stream via `System.err.println("I/O error");`.

**Note** `System.err` provides the same families of `println()` and `print()` methods as `System.out`. You should only switch from `System.out` to `System.err` when you need to output an error message so that the error messages are displayed on the screen, even when standard output is redirected to a file.

Compile Listing 1-3 via the following command line:

```
javac EchoText.java
```

Now run the application via the following command line:

```
java EchoText
```

You should see a flashing cursor. Type the following text and press Enter:

```
This is a test.
```

You should see this text duplicated on the following line and the application should end.

Continue by redirecting the input source to a file, by specifying the less than angle bracket (`<`) followed by a filename:

```
java EchoText <EchoText.java x
```

Although it looks like there are two command-line arguments, there is only one: x. (Redirection symbols followed by filenames don’t count as command-line arguments.) You should observe the contents of `EchoText.java` listed on the screen.

Finally, execute the following command line:

```
java EchoText <EchoText.java
```
This time, \( x \) isn’t specified, so input is assumed to originate from the keyboard. However, because the input is actually coming from the file `EchoText.java`, and because each line is terminated with a newline, only the first line from this file will be output.

**Note** If I had shortened the `while` loop expression to

\[
\text{while ((ch = System.in.read()) != -1)}
\]

and didn’t redirect standard input to a file, the loop wouldn’t end because `-1` would never be seen. To exit this loop, you would have to press the Ctrl and C keys simultaneously on a Windows platform or the equivalent keys on a non-Windows platform.

## Installing and Exploring the Eclipse IDE

Working with the JDK’s tools at the command line is probably okay for small projects. However, this practice isn’t recommended for large projects, which are hard to manage without the help of an IDE.

An **IDE** consists of a project manager for managing a project’s files, a text editor for entering and editing source code, a debugger for locating bugs, and other features. Eclipse is a popular IDE that Google supports for developing Android apps.

**Note** For convenience, I use JDK tools throughout this book, except for this section where I discuss and demonstrate the Eclipse IDE.

*Eclipse IDE* is an open source IDE for developing programs in Java and other languages (such as C, COBOL, PHP, Perl, and Python). Eclipse Standard is one distribution of this IDE that’s available for download; version 4.3.1 is the current version at time of this writing.

You should download and install Eclipse Standard to follow along with this section’s Eclipse-oriented example. Begin by pointing your browser to [www.eclipse.org/downloads/](http://www.eclipse.org/downloads/) and completing the following tasks.

1. Scroll down the page until you see an Eclipse Standard entry. (It may refer to 4.3.1 or a newer version.)
2. Click one of the platform links (such as Windows 64 Bit) to the right of this entry.
3. Select a download mirror from the subsequently displayed page, and proceed to download the distribution’s archive file.

I downloaded the `eclipse-standard-kepler-SR1-win32-x86_64.zip` archive file for my Windows 7 platform, unarchived this file, moved the resulting `eclipse` home directory to another location, and created a shortcut to that directory’s `eclipse.exe` file.

After installing Eclipse Classic, run this application. You should discover a splash screen identifying this IDE and a dialog box that lets you choose the location of a workspace for storing projects followed by a main window like the one shown in Figure 1-4.
Click the OK button, and you’re taken to Eclipse’s main window. See Figure 1-5.

Figure 1-5. The main window initially presents a Welcome tab
The main window initially presents a Welcome tab from which you can learn more about Eclipse. Click this tab’s X icon to close this tab; you can restore the Welcome tab by selecting Welcome from the menu bar’s Help menu.

The Eclipse user interface is based on a main window that consists of a menu bar, a tool bar, a workbench area, and a status bar. The workbench presents windows for organizing Eclipse projects, editing source files, viewing messages, and more.

To help you get comfortable with the Eclipse user interface, I’ll show you how to create a DumpArgs project containing a single DumpArgs.java source file with Listing 1-2’s source code. You’ll also learn how to compile and run this application.

Complete the following steps to create the DumpArgs project.

1. Select New from the File menu and Java Project from the resulting pop-up menu.

2. In the resulting New Java Project dialog box, enter DumpArgs into the Project name text field. Keep all of the other defaults, and click the Finish button.

After the second step (and after closing the Welcome tab), you’ll see a workbench similar to the one shown in Figure 1-6.
On the left side of the workbench, you’ll see a window titled Package Explorer. This window identifies the workspace’s projects in terms of packages (discussed in Chapter 5). At the moment, only a single DumpArgs entry appears in this window.

Clicking the triangle icon to the left of DumpArgs expands this entry to reveal src and JRE System Library items. The src item stores the DumpArgs project’s source files, and the JRE System Library item identifies various JRE files that are used to run this application.

You’ll now add a new file named DumpArgs.java to src.

1. Highlight src, and select New from the File menu and File from the resulting pop-up menu.

2. In the resulting New File dialog box, enter `DumpArgs.java` into the File name text field, and click the Finish button.

Eclipse responds by displaying an editor window titled DumpArgs.java. Copy Listing 1-2 content to this window. Then compile and run this application by selecting Run from the Run menu. (If you see a Save and Launch dialog box, click OK to close this dialog box.) Figure 1-7 shows the results.

![Figure 1-7](image-url)

*Figure 1-7. The Console tab at the bottom of the workbench presents the DumpArgs application’s output*
You must pass command-line arguments to `DumpArgs` to see additional output from this application. You can accomplish this task via these steps:

1. Select Run Configurations from the Run menu.
2. In the resulting Run Configurations dialog box, select the Arguments tab.
3. Enter `Curly Moe Larry` into the Program arguments text area, and click the Close button. See Figure 1-8.

Once again, select Run from the Run menu to run the `DumpArgs` application. This time, the Console tab reveals Curly, Moe, and Larry on separate lines below "Passed arguments:"

This is all I have to say about the Eclipse IDE. For more information, study the tutorials via the Welcome tab, access IDE help via the Help menu, and explore the Eclipse documentation at www.eclipse.org/documentation/.
Java Meets Android

In the previous two editions of this book, I provided an introduction to Java language features and assorted APIs that are helpful when developing Android apps. Apart from a few small references to various Android items, I didn’t delve into Android. This edition still explores Java language features and APIs that are useful in Android app development. However, it also introduces you to Android.

In this section, I first answer the “What is Android?” question. I next review Android’s history and its various releases. After exploring Android’s architecture, I present the Android version of HelloWorld.

Note  This isn’t all that I have to say about Android. In this book’s code archive (see the introduction for the details on how to obtain this freebie) I’ve included a PDF-based Appendix C that digs deeper into Android. I couldn’t include this information in the book proper because I only have room to present essential Java language features and APIs, which you must first learn.

What Is Android?

Android is Google’s software stack for mobile devices. This stack consists of apps (such as Browser and Contacts), a virtual machine in which apps run, middleware (software that sits on top of the operating system and provides various services to the virtual machine and its apps), and a Linux-based operating system.

Android offers the following features:

- An application framework enabling reuse and replacement of app components
- Bluetooth, EDGE, 3G, and Wi-Fi support (hardware dependent)
- Camera, GPS, compass, and accelerometer support (hardware dependent)
- Dalvik virtual machine optimized for mobile devices
- GSM Telephony support (hardware dependent)
- Integrated browser based on the open source WebKit engine
- Media support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- Optimized graphics powered by a custom 2D graphics library; 3D graphics based on OpenGL ES 1.0, 1.1, 2.0, or 3.0 (hardware acceleration optional)
- SQLite for structured data storage (I introduce SQLite in Chapter 14.)

Although not part of the software stack, Android’s rich development environment (including a device emulator and a plug-in for the Eclipse IDE) could also be considered an Android feature.
History of Android


On September 23, 2008, Google released Android 1.0, whose core features included a web browser, camera support, Google Search, Wi-Fi and Bluetooth support, and more. This release corresponds to API Level 1. (An API level is a 1-based integer that uniquely identifies the API revision offered by an Android version; it’s a way of distinguishing one significant Android release from another.)

Table 1-1 outlines subsequent releases. (Starting with version 1.5, each major release comes under a code name that’s based on a dessert item.)

<table>
<thead>
<tr>
<th>Version</th>
<th>API Level</th>
<th>Release Date and Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>2</td>
<td>Google released SDK 1.1 on February 9, 2009. Changes included showing/hiding the speakerphone dialpad and saving attachments in messages.</td>
</tr>
<tr>
<td>1.5 (Cupcake)</td>
<td>3</td>
<td>Google released SDK 1.5 on April 30, 2009. Changes included recording and watching videos in MPEG-4 and 3GP formats, populating the home screen (a special app that’s a starting point for using an Android device) with widgets (miniature app views), and animated screen transitions.</td>
</tr>
<tr>
<td>1.6 (Donut)</td>
<td>4</td>
<td>Google released SDK 1.6 on September 15, 2009. Changes included an expanded Gesture framework and the new GestureBuilder development tool, an integrated camera/camcorder/gallery interface, support for WVGA screen resolutions, and an updated search experience.</td>
</tr>
<tr>
<td>2.0 (Éclair)</td>
<td>5</td>
<td>Google released SDK 2.0 on October 26, 2009. Changes included live wallpapers, numerous new camera features (including flash support, digital zoom, scene mode, white balance, color effect, and macro focus), improved typing speed on the virtual keyboard, a smarter dictionary that learns from word usage and includes contact names as suggestions, improved Google Maps 3.1.2, and Bluetooth 2.1 support.</td>
</tr>
<tr>
<td>2.1 (Éclair)</td>
<td>7</td>
<td>Google released SDK update 2.1 on January 12, 2010. Version 2.1 presented minor amendments to the API and bug fixes.</td>
</tr>
</tbody>
</table>

(continued)
### Table 1-1. (continued)

<table>
<thead>
<tr>
<th>Version</th>
<th>API Level</th>
<th>Release Date and Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2 - 2.2.3 (Froyo)</td>
<td>8</td>
<td>Google released SDK 2.2 on May 20, 2009. Changes included the integration of Chrome’s V8 JavaScript engine into the Browser app, support for Bluetooth-enabled car and desk docks, Adobe Flash support, additional app speed improvements through JIT compilation, and USB tethering and Wi-Fi hotspot functionality. Google subsequently released SDK update 2.2.1 on January 18, 2011 to offer bug fixes, security updates, and performance improvements. It then released SDK update 2.2.2 on January 22, 2011 to provide minor bug fixes, including SMS routing issues that affected the Nexus One. Finally, Google released SDK update 2.2.3 on November 21, 2011 to provide two security patches.</td>
</tr>
<tr>
<td>2.3 - 2.3.2 (Gingerbread)</td>
<td>9</td>
<td>Google released SDK 2.3 on December 6, 2010. Changes included a new concurrent garbage collector that improves an app’s responsiveness, support for gyroscope and barometer sensing, support for WebM/VP8 video playback and AAC audio encoding, support for near field communication, and enhanced copy/paste functionality that lets users select a word by press-hold, copy, and paste. Google subsequently released SDK update 2.3.1 in December 2010 and SDK update 2.3.2 in January 2011. Both updates offered improvements and bug fixes for the Google Nexus S.</td>
</tr>
<tr>
<td>2.3.3 - 2.3.7 (Gingerbread)</td>
<td>10</td>
<td>Google released SDK update 2.3.3 on February 9, 2011, offering improvements and API fixes; SDK update 2.3.4 on April 28, 2011, adding support for voice or video chat via Google Talk and other features; SDK update 2.3.5 on July 25, 2011, offering camera software enhancements, shadow animations for list scrolling, improved battery efficiency, and more; SDK update 2.3.6 on September 2, 2011, fixing a voice search bug; and SDK update 2.3.7 on September 21, 2011, bringing support for Google Wallet to the Nexus S 4G.</td>
</tr>
<tr>
<td>3.0 (Honeycomb)</td>
<td>11</td>
<td>Google released SDK 3.0 on February 22, 2011. Unlike previous releases, version 3.0 focused exclusively on tablets, such as Motorola Xoom, the first tablet device featuring this version to be released. In addition to an improved and 3D user interface, version 3.0 improved multitasking, supported multicore processors, supported hardware acceleration, offered the ability to encrypt all user data, and more.</td>
</tr>
<tr>
<td>3.1 (Honeycomb)</td>
<td>12</td>
<td>Google released SDK 3.1 on May 10, 2011. Changes included user interface refinements, connectivity for USB accessories, support for joysticks and gamepads, and more.</td>
</tr>
<tr>
<td>Version</td>
<td>API Level</td>
<td>Release Date and Changes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.2 (Honeycomb)</td>
<td>13</td>
<td>Google released SDK 3.2 on July 15, 2011. Changes included improved hardware support, including optimizations for a wider range of tablets; a compatibility display mode for apps that haven’t been optimized for tablet screen resolutions; and more. Google subsequently released SDK updates 3.2.1, 3.2.2, 3.2.3, 3.2.4, 3.2.5, and 3.2.6 from September 2011 through February 2012.</td>
</tr>
<tr>
<td>4.0 - 4.0.2 (Ice Cream Sandwich)</td>
<td>14</td>
<td>Google released SDK 4.0.1 on October 19, 2011. SDK 4.0 unified the 2.3.x smartphone and 3.x tablet SDKs. Features included 1080p video recording, a customizable launcher, and more. Google subsequently released SDK updates 4.0.1 and 4.0.2 in late 2011 to fix bugs.</td>
</tr>
<tr>
<td>4.0.3 - 4.0.4 (Ice Cream Sandwich)</td>
<td>15</td>
<td>Google released SDK 4.0.3 on September 16, 2011. Changes included improvements to graphics, databases, spell-checking, and Bluetooth functionality; new APIs for developers, including a social stream API in the Contacts provider; calendar provider enhancements; new camera apps enhancing video stabilization and QVGA resolution; and accessibility refinements such as improved content access for screen readers. Google then released SDK 4.0.4 on March 29, 2012. Changes included stability improvements, better camera performance, smoother screen rotation, and improved phone number recognition.</td>
</tr>
<tr>
<td>4.1 (Jelly Bean)</td>
<td>16</td>
<td>Google released SDK 4.1 on July 9, 2012. Changes included vsync timing, triple buffering, automatically resizable app widgets, improved voice search, multichannel audio, and expandable notifications. Google subsequently released SDK 4.1.1 on July 23, 2012 to fix a bug on the Nexus 7 regarding the inability to change screen orientation in any application. It then released SDK 4.1.2 on October 9, 2012 to provide lock/home screen rotation support for the Nexus 7, one-finger gestures to expand/collapse notifications, and bug fixes and performance enhancements.</td>
</tr>
</tbody>
</table>

(continued)
Google released SDK 4.2 on November 13, 2012. Changes included “Photo Sphere” panorama photos; lock screen improvements; a new clock app with built-in world clock, stop watch and timer; support for wireless display (known as Miracast), and more.

Google subsequently released SDK 4.2.1 on November 27, 2012 to fix a bug in the People app where December wasn’t displayed on the date selector when adding an event to a contact and to add Bluetooth gamepads and joysticks as supported human interface devices. It then released SDK 4.2.2 on February 11, 2013 to fix Bluetooth audio streaming bugs; provide new download notifications, which now shows the percentage and estimated time remaining for active app downloads; provide new sounds for wireless charging and low battery; and more.

Google released SDK 4.3 on July 24, 2013. Changes included Bluetooth low-energy support; support for five more languages, improved digital rights management APIs, 4K resolution support, OpenGL ES 3.0 support (to allow for improved game graphics), and more.

Google subsequently released SDK 4.3.1 on October 3, 2013 to fix bugs and provide small tweaks for the Nexus 7 LTE.

Google released SDK 4.4 on October 31, 2013. Changes included streamlined memory usage and less heap usage, a loudness enhancer, screen recording, a transitions framework for animating scenes, a printing framework, full-screen immersive mode, audio monitoring, NFC Host Card Emulation, system-wide settings for closed captioning, and more.

Google subsequently released SDK update 4.4.1 on December 5, 2013 with some camera improvements, bug fixes, and more. It then released SDK update 4.4.2 on December 9, 2013 with some security enhancements, bug fixes, and the removal of the application permissions control system introduced in SDK 4.3.

Version 4.4.2 is the latest version at time of this writing. I focus on this version in Appendix C.

**Android Architecture**

The Android software stack consists of apps at the top, a Linux kernel with various drivers at the bottom, and middleware (an application framework, libraries, and the Android runtime) in the center. Figure 1-9 shows this layered architecture.
Users care very much about apps, and Android ships with a variety of useful core apps, which include Browser, Contacts, and Phone. All apps are written in the Java programming language. Apps form the top layer of Android’s architecture.

Android doesn’t officially recognize Java language features newer than Java 5, which is why I don’t discuss them in this book. Regarding APIs, this platform supports many APIs from Java 6 and previous Java versions. Also, Android provides its own unique APIs.

**Note** It’s possible to add support for Java language features that are more recent than Java 5 (see [www.informit.com/articles/article.aspx?p=1966024](http://www.informit.com/articles/article.aspx?p=1966024)).
Directly beneath the app layer is the application framework, a set of high-level building blocks for creating apps. The application framework is preinstalled on Android devices and consists of the following components:

- **Activity Manager**: This component provides an app's life cycle and maintains a shared activity stack for navigating within and among apps. Both topics are discussed in Appendix C.

- **Content Providers**: These components encapsulate data (such as the Browser app’s bookmarks) that can be shared among apps.

- **Location Manager**: This component makes it possible for an Android device to be aware of its physical location.

- **Notification Manager**: This component lets an app notify the user of a significant event (such as a message’s arrival) without interrupting what the user is currently doing.

- **Package Manager**: This component lets an app learn about other app packages that are currently installed on the device. (App packages are discussed in Appendix C.)

- **Resource Manager**: This component lets an app access its resources, a topic that’s discussed in Appendix C.

- **Telephony Manager**: This component lets an app learn about a device’s telephony services. It also handles making and receiving phone calls.

- **View System**: This component manages user interface elements and user interface-oriented event generation. (These topics are briefly discussed later in Appendix C.)

- **Window Manager**: This component organizes the screen’s real estate into windows, allocates drawing surfaces, and performs other window-related jobs.

The components of the application framework rely on a set of C/C++ libraries to perform their functions. Developers interact with the following libraries by way of framework APIs:

- **FreeType**: This library supports bitmap and vector font rendering.

- **libc**: This library is a BSD-derived implementation of the standard C system library, tuned for embedded Linux-based devices.

- **LibWebCore**: This library offers a modern and fast web browser engine that powers the Android browser and an embeddable web view. It's based on WebKit ([http://en.wikipedia.org/wiki/WebKit](http://en.wikipedia.org/wiki/WebKit)), and the Google Chrome and Apple Safari browsers also use it.

- **Media Framework**: These libraries, which are based on PacketVideo’s OpenCORE, support the playback and recording of many popular audio and video formats, as well as working with static image files. Supported formats include MPEG4, H.264, MP3, AAC, AMR, JPEG, PNG, and GIF.
- **OpenGL | ES**: These 3D graphics libraries provide an OpenGL implementation based on OpenGL ES 1.0/1.1/2.0/3.0 APIs. They use hardware 3D acceleration (where available) or the included (and highly optimized) 3D software rasterizer.

- **SGL**: This library provides the underlying 2D graphics engine.

- **SQLite**: This library provides a powerful and lightweight relational database engine that’s available to all apps and that’s also used by Mozilla Firefox and Apple’s iPhone for persistent storage.

- **SSL**: This library provides secure sockets layer-based security for network communication.

- **Surface Manager**: This library manages access to the display subsystem, and seamlessly composites 2D and 3D graphic layers from multiple apps.

Android provides a runtime environment that consists of core libraries (implementing a subset of the Apache Harmony Java version 5 implementation) and the *Dalvik virtual machine* (a non-JVM that’s based on processor registers instead of being stack-based).

**Note**  
Google’s Dan Bornstein created Dalvik and named this virtual machine after an Icelandic fishing village where some of his ancestors lived.

Each Android app defaults to running in its own Linux *process* (executing application), which hosts an instance of Dalvik. This virtual machine has been designed so that devices can run multiple virtual machines efficiently. This efficiency is largely due to Dalvik executing Dalvik Executable (DEX)-based files. DEX is a format that’s optimized for a minimal memory footprint.

**Note**  
Android starts a process when any part of the app needs to execute, and it shuts down the process when it’s no longer needed and system resources are required by other apps.

Perhaps you’re wondering how it’s possible to have a non-JVM run Java code. The answer is that Dalvik doesn’t run Java code. Instead, Android transforms compiled Java class files (see Figure 1-2) into the DEX format via its *dx* tool, and it’s this resulting code that gets executed by Dalvik.

Finally, the libraries and Android runtime rely on the Linux kernel (version 2.6.x, 3.0.x, or 3.8 [KitKat]) for underlying core services, such as threading, low-level memory management, a network stack, process management, and a driver model. Furthermore, the kernel acts as an abstraction layer between the hardware and the rest of the software stack.
Android’s architecture includes a security model that prevents apps from performing operations that are considered harmful to other apps, Linux, or users. This security model is mostly based on process level enforcement via standard Linux features (such as user and group IDs), and places processes in a security sandbox.

By default, the sandbox prevents apps from reading or writing the user’s private data (such as contacts or e-mails), reading or writing another app’s files, performing network access, keeping the device awake, accessing the camera, and so on. Apps that need to access the network or perform other sensitive operations must first obtain permission to do so.

Android handles permission requests in various ways, typically by automatically allowing or disallowing the request based upon a certificate or by prompting the user to grant or revoke the permission. Permissions required by an app are declared in the app’s manifest file (discussed in Appendix C) so that they are known to Android when the app is installed. These permissions won’t subsequently change.

**Android Says Hello**

Earlier in this chapter, I introduced you to HelloWorld, a Java application that outputs “Hello, World!” Because you might be curious about its Android equivalent, check out Listing 1-4.

**Listing 1-4. The Android Equivalent of HelloWorld**

```java
public class HelloWorld extends android.app.Activity
{
    public void create(android.os.Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        System.out.println("Hello, World!");
    }
}
```

Listing 1-4 isn’t too different from Listing 1-1, but there are some significant changes. For one thing, HelloWorld extends another class named Activity (stored in a package named android.app—see Chapter 5 for a discussion of packages). By extending Activity, HelloWorld proclaims itself as an activity, which you can think of as a user interface screen. (I discuss extension in Chapter 4.)

By extending Activity, HelloWorld inherits that class’s create() method, which Android calls when creating the activity. HelloWorld overrides this method with its own implementation so that it can output the “Hello, World!” message. However, create() first needs to execute Activity’s version of this method (via super.onCreate()), so that the activity is properly initialized.

**Note** “Hello, World!” isn’t displayed on an Android device’s screen. Instead, it’s written to a log file that can be examined by Android’s adb tool. Appendix C discusses the log file, adb, and writing to the screen.
That's enough for now. You'll learn more about `create()` and other life cycle methods in Appendix C. However, you first need to learn more about the Java language.

**Note**  HelloWorld, DumpArgs, and EchoText demonstrate `public static void main(String[] args)` as a Java application's entry point. This is where the application's execution begins. In contrast, as you've just seen, an Android app doesn’t require this method for its entry point because the app's architecture is very different.

---

**EXERCISES**

The following exercises are designed to test your understanding of Chapter 1’s content.

1. What is Java?
2. What is a virtual machine?
3. What is the purpose of the Java compiler?
4. True or false: A classfile’s instructions are commonly referred to as *bytecode*.
5. What does the JVM’s interpreter do when it learns that a sequence of bytecode instructions is being executed repeatedly?
6. How does the Java platform promote portability?
7. How does the Java platform promote security?
8. True or false: Java SE is the Java platform for developing servlets.
9. What is the JRE?
10. What is the difference between the public and private JREs?
11. What is the JDK?
12. Which JDK tool is used to compile Java source code?
13. Which JDK tool is used to run Java applications?
14. What is Standard I/O?
15. How do you specify the `main()` method's header?
16. What is an IDE? Identify the IDE that Google supports for developing Android apps.
17. What is Android?
18. What is the API level associated with Android 4.4?
19. What is the DEX format?
20. What tool does Android use to transform compiled Java classfiles into the DEX format?
Summary

Java is a language and a platform. The language is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers. The platform consists of a virtual machine and associated execution environment.

The Java language shares several similarities with C/C++, such as presenting the same single-line and multi-line comments and offering various reserved words that are also found in C/C++. However, there are differences, such as providing >>> and other operators not found in C/C++.

The Java platform includes a huge library of prebuilt classfiles that perform common tasks, such as math operations (trigonometry, for example) and network communications. This library is commonly referred to as the standard class library.

A special Java program known as the Java compiler translates source code into object code consisting of instructions that are executed by the JVM and associated data. These instructions are known as bytecode.

Developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices, and embedded devices. These editions are known as Java SE, Java EE, and Java ME.

The public JRE implements the Java SE platform and makes it possible to run Java programs. The JDK provides tools (including the Java compiler) for developing Java programs, and it also includes a private copy of the JRE.

Working with the JDK’s tools at the command line isn’t recommended for large projects, which are hard to manage without the help of an integrated development environment. Eclipse is a popular IDE that Google supports for developing Android apps.

Android is Google’s software stack for mobile devices. This stack consists of apps, a virtual machine in which apps run, middleware that sits on top of the operating system and provides various services to the virtual machine and its apps, and a Linux-based operating system.

Android didn’t originate with Google. Instead, Android, Inc., a small Palo Alto, California-based startup company, initially developed Android. Google bought this company in the summer of 2005, and it released Android 1.0 on September 23, 2008.

Android’s architecture is based on an application layer, an application framework, libraries, an Android runtime (consisting of core libraries implementing a subset of the Apache Harmony Java version 5 implementation and the Dalvik virtual machine), and a Linux kernel.

Android doesn’t officially recognize Java language features newer than Java 5, which is why I don’t discuss them in this book. Regarding APIs, this platform supports many APIs from Java 6 and previous Java versions. Also, Android provides its own unique APIs.

Chapter 2 introduces you to the Java language by focusing on this language’s fundamentals. You’ll learn about comments, identifiers, types, variables, expressions, statements, and more.
Learning Language Fundamentals

Aspiring Android app developers need to understand the Java language in which an app’s source code is written. This chapter introduces you to this language by focusing on its fundamentals. Specifically, you’ll learn about application structure, comments, identifiers (and reserved words), types, variables, expressions (and literals), and statements.

Note The American Standard Code for Information Interchange (ASCII) has traditionally been used to encode a program’s source code. Because ASCII is limited to the English language, Unicode (http://unicode.org/) was developed as a replacement. Unicode is a computing industry standard for consistently encoding, representing, and handling text that’s expressed in most of the world’s writing systems. Because Java supports Unicode, non-English-oriented symbols can be integrated into or accessed from Java source code; you’ll see examples in this chapter.

Learning Application Structure

Chapter 1 introduced you to three small Java applications. Each application exhibited a similar structure that I employ throughout this book. Before developing Java applications, you need to understand this structure, which Listing 2-1 presents. Throughout this chapter, I present code fragments that you can paste into this structure to create working applications.
Listing 2-1. Structuring a Java Application

```java
public class X {
    public static void main(String[] args) {
        ...
    }
}
```

An application is based on a class declaration. (I discuss classes in Chapter 3.) The declaration begins with a header consisting of `public`, followed by `class`, followed by `X`, where `X` is a placeholder for the actual name, for example, `HelloWorld`. The header is followed by a pair of braces `{ and }` that denote the class's body.

Between these braces is a special method declaration (I discuss methods in Chapter 3), which defines the application's entry point. It starts with a header that consists of `public`, followed by `static`, followed by `void`, followed by `main`, followed by `(String[] args)`. A pair of braces follows this header and denotes the method's body. The `...` represents code that you specify to execute.

You can pass a sequence of arguments to the application when executing it at the command line. These string-based arguments are stored in the `args` array (a `string` is a character sequence delimited by double quote `{"} characters). I introduce arrays later in this chapter and further discuss them in Chapter 3. There's nothing special about `args`: I could choose another name for it, for example, `arguments`.

You must store this class declaration in a file whose name matches `X` and has a `.java` file extension. You would then compile the source code as follows:

```
javac X.java
```

`X` is a placeholder for the actual class name. Also, the “.java” file extension is mandatory.

Assuming that compilation succeeds, which results in a classfile named `X.class` being created, you would subsequently run the application as follows:

```
java X
```

Replace `X` with the actual class name. Don’t specify the “.class” file extension.

If you need to pass command-line arguments to the application, specify them after the class name according to the following pattern:

```
java X arg1 arg2 arg3 ...
```

Here, `arg1`, `arg2`, and `arg3` are placeholders for three command-line arguments. The trailing `...` signifies additional arguments (if any).

Finally, if you need to specify a sequence of words as a single argument, place these words between double quotes to prevent `java` from treating them as separate arguments, like so:

```
java X "These words constitute a single argument."
```
CHAPTER 2: Learning Language Fundamentals

Learning Comments

Source code needs to be documented so that you (and any others who have to maintain it) can understand it, now and later. Source code should be documented while being written and whenever it's modified. If these modifications impact existing documentation, the documentation must be updated so that it accurately explains the code.

Java provides the comment feature for embedding documentation in source code. When the source code is compiled, the Java compiler ignores all comments; no bytecodes are generated. Single-line, multiline, and Javadoc comments are supported.

Single-Line Comments

A single-line comment occupies all or part of a single line of source code. This comment begins with the // character sequence and continues with explanatory text. The compiler ignores everything from // to the end of the line in which // appears.

The following example presents a single-line comment:

```
System.out.println(Math.sqrt(10 * 10 + 20 * 20));  // Output distance from (0, 0) to (10, 20).
```

This example calculates the distance between the (0, 0) origin and the point (10, 20) in the Cartesian x/y plane. It uses the formula \( \text{distance} = \text{square root}(x^2 + y^2) \), where \( x \) is 10 and \( y \) is 20, for this task. Java provides a Math class whose sqrt() method returns the square root of its single numeric argument. (I discuss Math in Chapter 7 and arguments in Chapter 3.)

Note Single-line comments are useful for inserting short but meaningful explanations of source code into this code. Don’t use them to insert unhelpful documentation. For example, when declaring a variable, don’t insert a meaningless comment such as // This variable stores integer values.

Multiline Comments

A multiline comment occupies one or more lines of source code. This comment begins with the /* character sequence, continues with explanatory text, and ends with the */ character sequence. Everything from /* through */ is ignored by the compiler.
The following example demonstrates a multiline comment:

```java
/*
   A year is a leap year when it's divisible by 400, or divisible by 4 and
   not also divisible by 100.
*/
System.out.println(year % 400 == 0 || (year % 4 == 0 && year % 100 != 0));
```

This example assumes the existence of an integer variable (discussed later) named `year` that stores an arbitrary four-digit year. It evaluates a complex expression (discussed later) that determines whether the year is leap or not. The expression returns true (leap year) or false (not leap year), which is output by `System.out.println()`. The multiline comment explains what constitutes a leap year.

Caution You cannot place one multiline comment inside another. For example, `/**/* Nesting multiline comments is illegal! */` isn't a valid multiline comment.

Javadoc Comments

Java supports a third kind of comment that simplifies the specification of external HTML-based documentation. You’ll find this Javadoc comment feature helpful in the preparation of technical documentation for other developers who rely on your Java applications, libraries, and other Java-based software products.

A Javadoc comment occupies one or more lines of source code. This comment begins with the `/**` character sequence, continues with explanatory text, and ends with the `*/` character sequence. Everything from `/**` through `*/` is ignored by the compiler.

The following example demonstrates a Javadoc comment:

```java
/**
   * Application entry point
   *
   * @param args array of command-line arguments passed to this method
   */
public static void main(String[] args)
{
   // TODO code application logic here
}
```

This example begins with a Javadoc comment that describes the `main()` method. Sandwiched between `/**` and `*/` is a description of the method and the `@param` Javadoc tag (an `@`-prefixed instruction to the javadoc tool).

The following list identifies several commonly used tags:

- `@author` identifies the source code’s author.
- `@deprecated` identifies a source code entity (such as a method) that should no longer be used.
@param identifies one of a method’s parameters.
@see provides a see-also reference.
@since identifies the software release where the entity first originated.
@return identifies the kind of value that the method returns.
@throws documents an exception thrown from a method. (I discuss exceptions in Chapter 5.)

Listing 2-2 presents Chapter 1’s DumpArgs application source code with Javadoc comments that describe the DumpArgs class and its main() method.

Listing 2-2. Documenting an Application Class and Its main() Method

```java
/**
   * Dump all command-line arguments to standard output.
   *
   * @author Jeff Friesen
   */

public class DumpArgs {
   /**
      * Application entry point.
      *
      * @param args array of command-line arguments.
      */
   public static void main(String[] args) {
      System.out.println("Passed arguments:");
      for (int i = 0; i < args.length; i++)
         System.out.println(args[i]);
   }
}
```

You can extract these documentation comments into a set of HTML files by using the JDK’s javadoc tool as follows:

javadoc DumpArgs.java

javadoc responds by outputting the following messages:

Loading source file DumpArgs.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_06
Building tree for all the packages and classes...
Generating \DumpArgs.html...
Generating \package-frame.html...
Generating \package-summary.html...
Generating \package-tree.html...
Generating \constant-values.html...
Building index for all the packages and classes...
Generating \overview-tree.html...
Generating \index-all.html...
Generating \deprecated-list.html...
Building index for all classes...
Generating \allclasses-frame.html...
Generating \allclasses-noframe.html...
Generating \index.html...
Generating \help-doc.html...

It also generates several files, including the index.html documentation entry-point file. Point your browser to this file, and you should see a page similar to that shown in Figure 2-1.

Figure 2-1. The entry-point page into DumpArg’s documentation describes this class
Learning Identifiers

Source code entities such as classes and methods need to be named so that they can be referenced from elsewhere in the code. Java provides the identifiers feature for this purpose.

An identifier consists of letters (A-Z, a-z, or equivalent uppercase/lowercase letters in other human alphabets), digits (0-9 or equivalent digits in other human alphabets), connecting punctuation characters (such as the underscore), and currency symbols (such as the dollar sign, $). This name must begin with a letter, a currency symbol, or a connecting punctuation character; and its length cannot exceed the line in which it appears.

Examples of valid identifiers include the following:

- $\pi$ (some editors might have problems with such symbols)
- i
- counter
- j2
- first$name
- _for

Examples of invalid identifiers include the following:

- 1name (starts with a digit)
- first#name (# isn’t a valid identifier symbol)

Note: Appendix B provides another (and a more extensive) example involving Javadoc comments and the javadoc tool.

Note: Java is a case-sensitive language, which means that identifiers differing only in case are considered separate identifiers. For example, temperature and Temperature are separate identifiers.

Almost any valid identifier can be chosen to name a class, method, or other source code entity. However, some identifiers are reserved for special purposes; they are known as reserved words. Java reserves the following identifiers:

- abstract
- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- false
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- null
- package
- private
- protected
- public
- return
- short
- static
The compiler outputs an error message when you attempt to use any of these reserved words outside of their usage contexts. Also, const and goto are not used by the Java language.

Listing 2-1 revealed several identifiers: public, class, X (a placeholder for an identifier), static, void, main, String, and args. Identifiers public, class, static, and void are also reserved words.

**Note** Most of Java's reserved words are also known as *keywords*. The three exceptions are false, null, and true, which are examples of *literals* (values specified verbatim).

### Learning Types

Applications process data items, such as integers, floating-point values, characters, and strings. Data items are classified according to various characteristics. For example, integers are whole numbers without fractions. Also, a string is a sequence of characters that's treated as a unit and possesses a length that identifies the number of characters in the sequence.

Java uses the term *type* to describe classifications of data items. A *type* identifies a set of data items (and their representation in memory) and a set of operations that transform these data items into other data items of that set. For example, the integer type identifies numeric values with no fractional parts and integer-oriented math operations, such as adding two integers to yield another integer.

**Note** Java is a *strongly typed language*, which means that every expression, variable, and so on has a type known to the compiler. This capability helps the compiler detect type-related errors at compile time rather than having these errors manifest themselves at runtime. Expressions and variables are discussed later in this chapter.

Java recognizes primitive types, user-defined types, and array types. These types are defined in the following sections.

### Primitive Types

A *primitive type* is a type that's defined by the language and whose values are not objects. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double precision floating-point primitive types. They are described in Table 2-1.
Table 2-1. Primitive Types

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Reserved Word</th>
<th>Size</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>boolean</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Character</td>
<td>char</td>
<td>16-bit</td>
<td>Unicode 0</td>
<td>Unicode 65,535</td>
</tr>
<tr>
<td>Byte integer</td>
<td>byte</td>
<td>8-bit</td>
<td>-128</td>
<td>+127</td>
</tr>
<tr>
<td>Short integer</td>
<td>short</td>
<td>16-bit</td>
<td>-32,768</td>
<td>+32,767</td>
</tr>
<tr>
<td>Integer</td>
<td>int</td>
<td>32-bit</td>
<td>-2,147,483,648</td>
<td>+2,147,483,647</td>
</tr>
<tr>
<td>Floating-point</td>
<td>float</td>
<td>32-bit</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
</tr>
<tr>
<td>Double precision floating-point</td>
<td>double</td>
<td>64-bit</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
</tr>
</tbody>
</table>

Table 2-1 describes each primitive type in terms of its reserved word, size, minimum value, and maximum value. A “--” entry indicates that the column in which it appears isn’t applicable to the primitive type described in that entry’s row.

The size column identifies the size of each primitive type in terms of the number of bits (binary digits; each digit is either 0 or 1) that a value of that type occupies in memory. Except for Boolean (whose size is implementation dependent; one Java implementation might store a Boolean value in a single bit, whereas another implementation might require an 8-bit byte for performance efficiency), each primitive type’s implementation has a specific size.

### BINARY VS. DECIMAL

Computers process numbers encoded via the binary number system, which is a base-2 number system in which there are only 2 digits: 0 and 1. In contrast, people process numbers according to the decimal number system, which is a base-10 number system in which there are 10 digits: 0 through 9.

It’s occasionally necessary to convert between binary integers and decimal integers. To convert from decimal to binary, you repeatedly follow these steps.

1. Set the integer quotient to the decimal integer.
2. If the quotient is 0, then stop.
3. Calculate quotient = quotient / 2. Output the remainder.
4. Go to Step 2.

For example, suppose you want to calculate the binary equivalent of decimal integer 19. According to the previous steps, you would need to perform these calculations:

19 / 2 = 9 Remainder 1
9 / 2 = 4 Remainder 1
4 / 2 = 2 Remainder 0
\[ \frac{2}{2} = 1 \text{ Remainder 0} \]
\[ \frac{1}{2} = 0 \text{ Remainder 1} \]

The first remainder in this list refers to the least significant digit of the resulting binary number, which is 10011 in this example.

To convert from binary to decimal, process the integer’s digits from right to left. Each of these digits represents a power of 2 where the rightmost digit's power is 0, the digit to its left has power 1, the digit to its left has power 2, and so on in an increasing sequence.

For example, 10011 corresponds to \( 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \). Because any number to the power 0 equals 1, this expression equates to \( 2^4 + 2^1 + 1 \), which equates to 16 + 2 + 1, which equals 19.

The minimum value and maximum value columns identify the smallest and largest values that can be represented by each type. Except for Boolean (whose only values are true and false), each primitive type has a minimum value and a maximum value.

The minimum and maximum values of the character type refer to Unicode. Unicode 0 is shorthand for “the first Unicode code point;” a code point is an integer that represents a symbol (such as A) or a control character (such as newline or tab) or that combines with other code points to form a symbol.

**Note** The character type’s limits imply that this type is **unsigned** (all character values are positive). In contrast, each numeric type is **signed** (it supports positive and negative values).

The minimum and maximum values of the byte integer, short integer, integer, and long integer types reveal that there is one more negative value than positive value (0 is typically not regarded as a positive value). The reason for this imbalance has to do with how integers are represented.

Java represents an integer value as a combination of a **sign bit** (the leftmost bit; 0 for a positive value and 1 for a negative value) and **magnitude bits** (all remaining bits to the right of the sign bit). When the sign bit is 0, the magnitude is stored directly. However, when the sign bit is 1, the magnitude is stored using **twos-complement representation** in which all 1s are flipped to 0s, all 0s are flipped to 1s, and 1 is added to the number behind the minus sign.

Twos-complement is used so that negative integers can naturally coexist with positive integers. For example, adding the representation of -1 to +1 yields 0. Figure 2-2 illustrates byte integer 2’s direct representation and byte integer -2’s twos-complement representation.

![Figure 2-2. The binary representation of two byte-integer values begins with a sign bit](image-url)
The minimum and maximum values of the floating-point and double precision floating-point types refer to Institute of Electrical and Electronics Engineers (IEEE) 754, which is a standard for representing floating-point values in memory. Check out Wikipedia’s “IEEE 754-2008” entry (http://en.wikipedia.org/wiki/IEEE_754) to learn more about this standard.

**Note**  Developers who argue that Java should support objects only aren’t happy about the inclusion of primitive types in the language. However, Java was designed to include primitive types to overcome the speed and memory limitations of early 1990s-era devices, to which Java was originally targeted.

### User-Defined Types

A **user-defined type** is a type that’s often used to model a real-world concept (such as a color or a bank account). The developer defines it using a class, an interface, an enum, or an annotation type; and its values are objects. (I discuss classes in Chapter 3, interfaces in Chapter 4, and enums and annotation types in Chapter 6.)

For example, you could create a Color class to model colors; its values could describe colors as red/green/blue components and its methods (see Chapter 3) could return these components.

**Note**  You can think of Chapter 1’s HelloWorld, DumpArgs, and EchoText classes as examples of user-defined types. However, these classes aren’t used to create objects but describe applications instead.

Java’s String class defines the string user-defined type and is a member of the standard class library. Its values describe character sequences, and its methods perform string operations such as joining two strings. Unlike other user-defined types, String enjoys language support for initializing String variables and joining strings into a single string. You’ll see examples of this later in the chapter.

User-defined types are also known as **reference types** because a variable of that type stores a reference (a memory address or some other identifier) to a region of memory that stores an object of that type. In contrast, variables of primitive types store the values directly; they don’t store references to these values.

### Array Types

An **array type** is a special reference type that signifies an array, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as elements. This type consists of the **element type** (a primitive type, user-defined type, or array type) and one or more pairs of square brackets that indicate the number of dimensions (extents). A single pair of brackets signifies a one-dimensional array (a vector), two pairs of brackets signify a two-dimensional array (a table), three pairs of brackets signify a one-dimensional array of two-dimensional arrays (a vector of tables), and so on. For example, int[] signifies a one-dimensional array (with int as the element type), and double[][][] signifies a two-dimensional array (with double as the element type).
CHAPTER 2: Learning Language Fundamentals

Learning Variables

Applications manipulate values that are stored in memory, which is symbolically represented in source code through the use of the variables feature. A variable is a named memory location that stores some type of value. A variable that stores a reference is often referred to as a reference variable.

Variables must be declared before they’re used. A declaration minimally consists of a type name, optionally followed by a sequence of square bracket pairs, followed by a name, optionally followed by a sequence of square bracket pairs, and terminated with a semicolon character (;). Consider the following examples:

```java
int counter;         // Declare integer variable counter.
double temperature; // Declare double precision floating-point variable temperature.
String firstName;    // Declare String variable firstName.
int[] ages;          // Declare one-dimensional integer array variable ages.
char gradeLetters[]; // Declare one-dimensional character array variable gradeLetters.
float[][] matrix;    // Declare two-dimensional floating-point array variable matrix.
double p;            // Declare double precision floating-point variable p.
```

No string is yet associated with firstName, and no arrays are yet associated with ages, gradeLetters, and matrix.

Note: Square brackets can appear after the type name or after the variable name, but not in both places. For example, the compiler reports an error when it encounters `int[] x[];`. It is common practice to place the square brackets after the type name (as in `int[] ages;`) instead of after the variable name (as in `char gradeLetters[];`), unless the array is being declared in a context such as `int x, y[], z;`.

You can declare multiple variables on one line by separating each variable from its predecessor with a comma, as demonstrated by the following example:

```java
int x, y[], z;
```

This example declares three variables named x, y, and z. Each variable shares the same type, which happens to be integer. Unlike x and z, which store single integer values, y[] signifies a one-dimensional array whose element type is integer; each element stores an integer value. No array is yet associated with y.

The square brackets must appear after the variable name when the array is declared on the same line as the other variables. If you place the square brackets before the variable name, as in `int x, []y, z;`, the compiler reports an error. If you place the square brackets after the type name, as in `int[] x, y, z;`, all three variables signify one-dimensional arrays of integers.
Learning Expressions

The previously declared variables were not explicitly initialized to any values. As a result, they are either initialized to default values (such as 0 for int and 0.0 for double) or remain uninitialized, depending on the contexts in which they appear (declared within classes or declared within methods). In Chapter 3, I discuss variable contexts in terms of local variables, parameters, and fields.

Java provides the expressions feature for initializing variables and for other purposes. An expression is a combination of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression’s type. If the expression is being assigned to a variable, this type must agree with the variable’s type; otherwise, the compiler reports an error.

Java recognizes simple expressions and compound expressions. These types are defined in the following sections.

Simple Expressions

A simple expression is a literal (a value expressed verbatim), the name of a variable (containing a value), or a method call (returning a value). Java supports several kinds of literals: string, Boolean true and false, character, integer, floating-point, and null.

Note  A method call that doesn’t return a value—the called method is known as a void method—is a special kind of simple expression, for example, System.out.println("Hello, World!"); This standalone expression cannot be assigned to a variable. Attempting to do so (as in int i = System.out.println("X")) causes the compiler to report an error.

A string literal consists of a sequence of Unicode characters surrounded by a pair of double quotes, for example, "The quick brown fox jumps over the lazy dog." It might also contain escape sequences, which are special syntax for representing certain printable and nonprintable characters that cannot otherwise appear in the literal. For example, "The quick brown "fox" jumps over the lazy dog." uses the " escape sequence to surround fox with double quotes.

Table 2-2 describes all supported escape sequences.
Finally, a string literal might contain Unicode escape sequences, which are special syntax for representing Unicode characters. A Unicode escape sequence begins with \u and continues with four hexadecimal digits (0-9, A-F, a-f) with no intervening space. For example, \u0041 represents capital letter A, and \u20ac represents the European Union’s euro currency symbol.

A Boolean literal consists of reserved word `true` or reserved word `false`.

A character literal consists of a single Unicode character surrounded by a pair of single quotes (‘A’ is an example). You can also represent, as a character literal, an escape sequence (‘\ ‘, for example) or a Unicode escape sequence (such as ‘\u0041’).

An integer literal consists of a sequence of digits. If the literal is to represent a long integer value, it must be suffixed with an uppercase L or lowercase l (L is easier to read). If there is no suffix, the literal represents a 32-bit integer (an int).

Integer literals can be specified in the decimal, hexadecimal, and octal formats:

- The decimal format is the default format, for example, 127.
- The hexadecimal format requires that the literal begin with 0x or 0X and continue with hexadecimal digits (0-9, A-F, a-f), for example, 0x7F.
- The octal format requires that the literal be prefixed with 0 and continue with octal digits (0-7), for example, 0177.

A floating-point literal consists of an integer part, a decimal point (represented by the period [.]), a fractional part, an exponent (starting with letter E or e), and a type suffix (letter D, d, F, or f). Most parts are optional, but enough information must be present to differentiate the floating-point literal from an integer literal. Examples include 0.1 (double precision floating-point), 89F (floating-point), 600D (double precision floating-point), and 13.08E+23 (double precision floating-point).

Finally, the null literal is assigned to a reference variable to indicate that the variable doesn’t refer to an object.

Listing 2-3 presents a SimpleExpressions application that uses literals to initialize the previously presented variables.
Listing 2-3. Using Literals to Initialize Variables

```java
public class SimpleExpressions {
    public static void main(String[] args) {
        int counter = 10;
        double temperature = 98.6; // Assume Fahrenheit scale.
        String firstName = "Mark";
        int[] ages = { 52, 28, 93, 16 };
        char gradeLetters[] = { 'A', 'B', 'C', 'D', 'F' };
        float[][] matrix = { { 1.0F, 2.0F, 3.0F }, { 4.0F, 5.0F, 6.0F } };
        int x = 1, y[] = { 1, 2, 3 }, z = 3;
        double π = 3.14159;
        System.out.println(counter);
        System.out.println(temperature);
        System.out.println(ages.length);
        System.out.println(gradeLetters.length);
        System.out.println(matrix.length);
        System.out.println(x);
        System.out.println(y.length);
        System.out.println(z);
        System.out.println(π);
    }
}
```

The first example assigns 32-bit integer literal 10 to 32-bit integer variable counter. The second example assigns double precision floating-point literal 98.6 to double precision floating-point variable temperature. The third example assigns string literal "Mark" to String variable firstName.

The fourth through seventh examples use array initializers (such as { 52, 28, 93, 16 }) to initialize arrays that are assigned to the ages, gradeLetters, matrix, and y array variables. An array initializer consists of a brace-and-comma-delimited list of expressions, which (as the matrix example shows) may be array initializers. The matrix example results in a table that looks like the following:

```
1.0F 2.0F 3.0F
4.0F 5.0F 6.0F
```

Each array variable is associated with a .length property that returns the number of elements in the array. For example, because ages contains 4 elements, ages.length returns 4. Similarly, because matrix contains 2 rows, matrix.length returns 2. I'll have more to say about this property and also show you how to access array elements later in this chapter.

When you attempt to save this listing using an editor such as Windows Notepad, you'll probably be prompted to change the encoding from extended ASCII to Unicode (unless you've previously done so); otherwise, the π symbol will be lost. To compile the saved source code, you'll then need to include the -encoding Unicode option, as follows:

```
javac -encoding Unicode SimpleExpressions.java
```
You can then run this application via the following command line:

```
java SimpleExpressions
```

You should observe the following output:

```
10
98.6
4
5
2
1
3
3
3.14159
```

### ORGANIZING VARIABLES IN MEMORY

Perhaps you’re curious about how variables are organized in memory. Figure 2-3 presents one possible high-level organization for the `counter`, `ages`, and `matrix` variables, along with the arrays assigned to `ages` and `matrix`.

![Figure 2-3. The counter variable stores a 4-byte integer value, whereas ages and matrix store 4-byte references to their respective arrays](image)

Figure 2-3 reveals that each of `counter`, `ages`, and `matrix` is stored at a memory address (starting at a fictitious `20001000` value in this example) that’s divisible by 4 (each variable stores a 4-byte value); that `counter`’s 4-byte value is stored at this address; and that each of the `ages` and `matrix` 4-byte memory locations stores the 32-bit address of its respective array (64-bit addresses would most likely be used on 64-bit virtual machines). Also, a one-dimensional array is stored as a list of values, whereas a two-dimensional array is stored as a one-dimensional row array of addresses, where each address identifies a one-dimensional column array of values for that row.

Although Figure 2-3 implies that array addresses are stored in `ages` and `matrix`, which equates references with addresses, a Java implementation might equate references with *handles* (integer values that identify slots in a list). This alternative is presented in Figure 2-4 for `ages` and its referenced array.
Handles make it easy to move around regions of memory during garbage collection (discussed in Chapter 3). If multiple variables referenced the same array via the same address, each variable’s address value would have to be updated when the array was moved. However, if multiple variables referenced the array via the same handle, only the handle’s list entry would need to be updated. A downside to using handles is that accessing memory via these handles can be slower than directly accessing this memory via an address. Regardless of how references are implemented, this implementation detail is hidden from the Java developer to promote portability.

In addition to assigning literals to variables, you can also assign variables and the results of method calls to variables, like so:

```java
int counter2 = counter;            // Assign previous counter variable value to counter2.
boolean isLeap = isLeapYear(2012); // Assign Boolean result of calling isLeapYear(2012) to isLeap.
```

These examples have assumed that only those expressions whose types are the same as the types of the variables that they are initializing can be assigned to those variables. However, under certain circumstances, it’s possible to assign an expression having a different type. For example, Java permits you to assign certain integer literals to short integer variables, as in `short s = 20;`, and assign a short integer expression to an integer variable, as in `int i = s;`.

Java permits the former assignment because 20 can be represented as a short integer (no information is lost). In contrast, Java would complain about `short s = 40000;` because integer literal 40000 cannot be represented as a short integer (32767 is the maximum positive integer that can be stored in a short integer variable). Java permits the latter assignment because no information is lost when Java converts from a type with a smaller set of values to a type with a wider set of values.

Java supports the following primitive-type conversions via widening conversion rules:

- Byte integer to short integer, integer, long integer, floating-point, or double precision floating-point
- Short integer to integer, long integer, floating-point, or double precision floating-point
- Character to integer, long integer, floating-point, or double precision floating-point
- Integer to long integer, floating-point, or double precision floating-point
- Long integer to floating-point or double precision floating-point
- Floating-point to double precision floating-point
Listing 2-4 presents a SimpleExpressions application that demonstrates these additional insights into simple expressions.

**Listing 2-4. Learning More About Simple Expressions**

```java
public class SimpleExpressions {
    public static void main(String[] args) {
        int counter = 30;
        int counter2 = counter;
        System.out.println(counter);

        short s = 20;
        System.out.println(s);
        int i = s;
        System.out.println(i);

        // short s2 = 40000; // possible loss of precision error
        int i2 = -1;
        double d = i2;
        System.out.println(d);
    }
}
```

This application demonstrates assigning one variable to another and assigning literal values to variables where the types don’t match. For example, in the `double d = i2;` example, a widening conversion rule converts the 32-bit integer value stored in variable `i2` to a double precision floating-point value that’s assigned to variable `d`.

Compile Listing 2-4 as follows:

`javac SimpleExpressions.java`

Unlike in the previous SimpleExpressions application, it isn’t necessary to specifying `-encoding Unicode` because this listing contains no characters apart from those characters that can be represented by the extended ASCII character set (which is a subset of Unicode).

Run this application via the following command line:

`java SimpleExpressions`

You should observe the following output:

```
30
20
20
-1.0
```
Note When converting from a smaller integer to a larger integer, Java copies the smaller integer’s sign bit into the extra bits of the larger integer.

In Chapter 4, I discuss the widening conversion rules for performing type conversions in the contexts of user-defined and array types.

**Compound Expressions**

A *compound expression* is a sequence of simple expressions and operators, where an *operator* (a sequence of instructions symbolically represented in source code) transforms its *operand* expression value(s) into another value. For example, -6 is a compound expression consisting of operator - and integer literal 6 as its operand. This expression transforms 6 into its negative equivalent. Similarly, x + 5 is a compound expression consisting of variable name x, integer literal 5, and operator + sandwiched between these operands. Variable x’s value is fetched and added to 5 when this expression is evaluated. The sum becomes the value of the expression.

Note When x’s type is byte integer or short integer, this variable’s value is widened to an integer. However, when x’s type is long integer, floating-point, or double precision floating-point, 5 is widened to the appropriate type. The addition operation is performed after the widening conversion takes place.

Java supplies many operators, which are classified by the number of operands that they take. A *unary operator* takes only one operand (unary minus [-] is an example), a *binary operator* takes two operands (addition [+ ] is an example), and Java’s single *ternary operator* (conditional [? :) ] takes three operands.

Operators are also classified as prefix, postfix, and infix. A *prefix operator* is a unary operator that precedes its operand (as in -6), a *postfix operator* is a unary operator that trails its operand (as in x++), and an *infix operator* is a binary or ternary operator sandwiched between the binary operator’s two or the ternary operator’s three operands (as in x + 5).

Table 2-3 presents all supported operators in terms of their symbols, descriptions, and precedence levels; I discuss the concept of precedence at the end of this section. Various operator descriptions refer to “integer type,” which is shorthand for specifying any of byte integer, short integer, integer, or long integer unless “integer type” is qualified as a 32-bit integer. Also, “numeric type” refers to any of these integer types along with floating-point and double precision floating-point.
Table 2-3. Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>Given ( \text{operand1} + \text{operand2} ), where each operand must be of character or numeric type, add \text{operand2} to \text{operand1} and return the sum.</td>
<td>10</td>
</tr>
<tr>
<td>Array index</td>
<td>[ ]</td>
<td>Given ( \text{variable}[\text{index}] ), where \text{index} must be of integer type, read value from or store value into \text{variable}'s storage element at location \text{index}.</td>
<td>13</td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td>Given ( \text{variable} = \text{operand} ), which must be assignment-compatible (their types must agree), store \text{operand} in \text{variable}.</td>
<td>0</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>Given ( \text{operand1} &amp; \text{operand2} ), where each operand must be of character or integer type, bitwise AND their corresponding bits and return the result. A result bit is set to 1 when each operand’s corresponding bit is 1. Otherwise, the result bit is set to 0.</td>
<td>6</td>
</tr>
<tr>
<td>Bitwise complement</td>
<td>~</td>
<td>Given ( ^{\text{operand}} ), where \text{operand} must be of character or integer type, flip \text{operand}'s bits (1s to 0s and 0s to 1s) and return the result.</td>
<td>12</td>
</tr>
<tr>
<td>Bitwise exclusive OR</td>
<td>^</td>
<td>Given ( \text{operand1} ^{\text{operand2}} ), where each operand must be of character or integer type, bitwise exclusive OR their corresponding bits and return the result. A result bit is set to 1 when one operand’s corresponding bit is 1 and the other operand’s corresponding bit is 0. Otherwise, the result bit is set to 0.</td>
<td>5</td>
</tr>
<tr>
<td>Bitwise inclusive OR</td>
<td></td>
<td>Given ( \text{operand1} \mid \text{operand2} ), which must be of character or integer type, bitwise inclusive OR their corresponding bits and return the result. A result bit is set to 1 when either (or both) of the operands’ corresponding bits is 1. Otherwise, the result bit is set to 0.</td>
<td>4</td>
</tr>
<tr>
<td>Cast</td>
<td>(type)</td>
<td>Given ( \text{(type) operand} ), convert \text{operand} to an equivalent value that can be represented by \text{type}. For example, you could use this operator to convert a floating-point value to a 32–bit integer value.</td>
<td>12</td>
</tr>
<tr>
<td>Compound</td>
<td>+=, -=, *=, /=, %=, &amp;=,</td>
<td>=, ^=, &lt;&lt;=, &gt;&gt;&gt;=</td>
<td>Given ( \text{variable operator operand} ), where \text{operator} is one of the listed compound operator symbols and where \text{operand} is assignment-compatible with \text{variable}, perform the indicated operation using \text{variable}'s value as \text{operator}'s left operand value and store the resulting value in \text{variable}.</td>
</tr>
<tr>
<td>assignment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>? :</td>
<td>Given ( \text{operand1} ? \text{operand2 : operand3} ), where \text{operand1} must be of Boolean type, return \text{operand2} when \text{operand1} is true or \text{operand3} when \text{operand1} is false. The types of \text{operand2} and \text{operand3} must agree.</td>
<td>1</td>
</tr>
</tbody>
</table>

(continued)
Table 2-3. (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional AND</td>
<td>&amp;&amp;</td>
<td>Given <em>operand1</em> &amp;&amp; <em>operand2</em>, where each operand must be of Boolean type, return true when both operands are true. Otherwise, return false. When <em>operand1</em> is false, <em>operand2</em> isn’t examined. This is known as short-circuiting.</td>
<td>3</td>
</tr>
<tr>
<td>Conditional OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>Given <em>operand1</em> / <em>operand2</em>, where each operand must be of character or numeric type, divide <em>operand1</em> by <em>operand2</em> and return the quotient.</td>
<td>11</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>Given <em>operand1</em> == <em>operand2</em>, where both operands must be comparable (you cannot compare an integer with a string literal, for example), compare both operands for equality. Return true when these operands are equal. Otherwise, return false.</td>
<td>7</td>
</tr>
<tr>
<td>Inequality</td>
<td>!=</td>
<td>Given <em>operand1</em> != <em>operand2</em>, where both operands must be comparable (you cannot compare an integer with a Boolean value, for example), compare both operands for inequality. Return true when these operands are not equal. Otherwise, return false.</td>
<td>7</td>
</tr>
<tr>
<td>Left shift</td>
<td>&lt;&lt;</td>
<td>Given <em>operand1</em> &lt;&lt; <em>operand2</em>, where each operand must be of character or integer type, shift <em>operand1</em>’s binary representation left by the number of bits that <em>operand2</em> specifies. For each shift, a 0 is shifted into the rightmost bit and the leftmost bit is discarded. Only the 5 low-order bits of <em>operand2</em> are used when shifting a 32–bit integer (to prevent shifting more than the number of bits in a 32–bit integer). Only the 6 low-order bits of <em>operand2</em> are used when shifting a 64-bit integer (to prevent shifting more than the number of bits in a 64-bit integer). The shift preserves negative values. Furthermore, it’s equivalent to (but faster than) multiplying by a multiple of 2.</td>
<td>9</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;</td>
<td>Given <em>operand1</em> &amp; <em>operand2</em>, where each operand must be of Boolean type, return true when both operands are true. Otherwise, return false. In contrast to conditional AND, logical AND doesn’t perform short-circuiting.</td>
<td>6</td>
</tr>
<tr>
<td>Logical complement</td>
<td>!</td>
<td>Given !<em>operand</em>, where <em>operand</em> must be of Boolean type, flip <em>operand</em>’s value (true to false or false to true) and return the result.</td>
<td>12</td>
</tr>
</tbody>
</table>

(continued)
### Table 2-3. (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical exclusive OR</td>
<td>^</td>
<td>Given operand1 ^ operand2, where each operand must be of Boolean type, return true when one operand is true and the other operand is false. Otherwise, return false.</td>
<td>5</td>
</tr>
<tr>
<td>Logical inclusive OR</td>
<td></td>
<td>Given operand1</td>
<td>operand2, where each operand must be of Boolean type, return true when at least one operand is true. Otherwise, return false. In contrast to conditional OR, logical inclusive OR doesn’t perform short-circuiting.</td>
</tr>
<tr>
<td>Member access</td>
<td>.</td>
<td>Given identifier1.identifier2, access the identifier2 member of identifier1. You’ll learn about this operator in Chapter 3.</td>
<td>13</td>
</tr>
<tr>
<td>Method call</td>
<td>()</td>
<td>Given identifier(argument list), call the method identified by identifier and matching parameter list. You’ll learn about method calling in Chapter 3.</td>
<td>13</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>Given operand1 * operand2, where each operand must be of character or numeric type, multiply operand1 by operand2 and return the product.</td>
<td>11</td>
</tr>
<tr>
<td>Object creation</td>
<td>new</td>
<td>Given new identifier(argument list), allocate memory for object and call constructor (discussed in Chapter 3) specified as identifier(argument list). Given new identifier[integer size], allocate a one-dimensional array of values.</td>
<td>12</td>
</tr>
<tr>
<td>Postdecrement</td>
<td>--</td>
<td>Given variable--, where variable must be of character or numeric type, subtract 1 from variable’s value (storing the result in variable) and return the original value.</td>
<td>13</td>
</tr>
<tr>
<td>Postincrement</td>
<td>++</td>
<td>Given variable++, where variable must be of character or numeric type, add 1 to variable’s value (storing the result in variable) and return the original value.</td>
<td>13</td>
</tr>
<tr>
<td>Predecrement</td>
<td>--</td>
<td>Given --variable, where variable must be of character or numeric type, subtract 1 from its value, store the result in variable, and return the new decremented value.</td>
<td>12</td>
</tr>
<tr>
<td>Preincrement</td>
<td>++</td>
<td>Given ++variable, where variable must be of character or numeric type, add 1 to its value, store the result in variable, and return the new incremented value.</td>
<td>12</td>
</tr>
<tr>
<td>Relational greater than</td>
<td>&gt;</td>
<td>Given operand1 &gt; operand2, where each operand must be of character or numeric type, return true when operand1 is greater than operand2. Otherwise, return false.</td>
<td>8</td>
</tr>
<tr>
<td>Relational greater than or equal to</td>
<td>&gt;=</td>
<td>Given operand1 &gt;= operand2, where each operand must be of character or numeric type, return true when operand1 is greater than or equal to operand2. Otherwise, return false.</td>
<td>8</td>
</tr>
<tr>
<td>Operator</td>
<td>Symbol</td>
<td>Description</td>
<td>Precedence</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Relational less than</td>
<td>&lt;</td>
<td>Given ( operand_1 &lt; operand_2 ), where each operand must be of character or numeric type, return true when ( operand_1 ) is less than ( operand_2 ). Otherwise, return false.</td>
<td>8</td>
</tr>
<tr>
<td>Relational less than or equal to</td>
<td>&lt;=</td>
<td>Given ( operand_1 \leq operand_2 ), where each operand must be of character or numeric type, return true when ( operand_1 ) is less than or equal to ( operand_2 ). Otherwise, return false.</td>
<td>8</td>
</tr>
<tr>
<td>Relational type checking</td>
<td>instanceof</td>
<td>Given ( operand_1 ) instanceof ( operand_2 ), where ( operand_1 ) is an object and ( operand_2 ) is a class (or other user-defined type) return true when ( operand_1 ) is an instance of ( operand_2 ). Otherwise, return false.</td>
<td>8</td>
</tr>
<tr>
<td>Remainder</td>
<td>%</td>
<td>Given ( operand_1 % operand_2 ), where each operand must be of character or numeric type, divide ( operand_1 ) by ( operand_2 ) and return the remainder. Also known as the modulus operator.</td>
<td>11</td>
</tr>
<tr>
<td>Signed right shift</td>
<td>&gt;&gt;</td>
<td>Given ( operand_1 \gg operand_2 ), where each operand must be of character or integer type, shift ( operand_1 )'s binary representation right by the number of bits that ( operand_2 ) specifies. For each shift, a copy of the sign bit (the leftmost bit) is shifted to the right and the rightmost bit is discarded. Only the 5 low-order bits of ( operand_2 ) are used when shifting a 32-bit integer (to prevent shifting more than the number of bits in a 32-bit integer). Only the 6 low-order bits of ( operand_2 ) are used when shifting a 64-bit integer (to prevent shifting more than the number of bits in a 64-bit integer). The shift preserves negative values. Furthermore, it's equivalent to (but faster than) dividing by a multiple of 2.</td>
<td>9</td>
</tr>
<tr>
<td>String concatenation</td>
<td>+</td>
<td>Given ( operand_1 + operand_2 ), where at least one operand is of String type, append ( operand_2 )'s string representation to ( operand_1 )'s string representation and return the concatenated result.</td>
<td>10</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>Given ( operand_1 - operand_2 ), where each operand must be of character or numeric type, subtract ( operand_2 ) from ( operand_1 ) and return the difference.</td>
<td>10</td>
</tr>
<tr>
<td>Unary minus</td>
<td>-</td>
<td>Given ( -operand ), where ( operand ) must be of character or numeric type, return ( operand )'s arithmetic negative.</td>
<td>12</td>
</tr>
<tr>
<td>Unary plus</td>
<td>+</td>
<td>Like its predecessor, but return ( operand ). Rarely used.</td>
<td>12</td>
</tr>
</tbody>
</table>

*Table 2-3. (continued)*
Table 2-3’s operators can be classified as additive, array index, assignment, bitwise, cast, conditional, equality, logical, member access, method call, multiplicative, object creation, relational, shift, and unary minus/plus.

### Additive Operators

The additive operators consist of addition (+), subtraction (-), postdecrement (- -), postincrement (+ +), predecrement (- -), preincrement (+ +), and string concatenation (+). Addition returns the sum of its operands (such as 6 + 4 returns 10), subtraction returns the difference between its operands (such as 6 - 4 returns 2 and 4 - 6 returns -2), postdecrement subtracts 1 from its variable operand and returns the variable’s prior value (such as x--), postincrement adds 1 to its variable operand and returns the variable’s prior value (such as x++), predecrement subtracts 1 from its variable operand and returns the variable’s new value (such as --x), preincrement adds 1 to its variable operand and returns the variable’s new value (such as ++x), and string concatenation merges its string operands and returns the merged string (such as "A" + "B" returns "AB").

The addition, subtraction, postdecrement, postincrement, predecrement, and preincrement operators can yield values that overflow or underflow the limits of the resulting value’s type. For example, adding two large positive 32-bit integer values can produce a value that cannot be represented as a 32-bit integer value. The result is said to overflow. Java doesn’t detect overflows and underflows.

Java provides a special widening conversion rule for use with string operands and the string concatenation operator. When either operand isn’t a string, the operand is converted to a string prior to string concatenation. For example, when presented with "A" + 5, the compiler generates code that first converts 5 to "5" and then performs the string concatenation operation, resulting in "A5".

Listing 2-5 presents a CompoundExpressions application that lets you start experimenting with the additive operators.
Listing 2-5. Experimenting with the Additive Operators

```java
public class CompoundExpressions {
    public static void main(String[] args) {
        int age = 65;
        System.out.println(age + 32);
        System.out.println(++age);
        System.out.println(age--);
        System.out.println("A" + "B");
        System.out.println("A" + 5);
        short x = 32767;
        System.out.println(++x);
    }
}
```

Listing 2-5’s main() method first declares a 32-bit integer age variable that’s initialized to 32-bit integer value 65. It then outputs the result of an expression that adds age’s value to 32-bit integer value 32.

The preincrement and postdecrement operators are now demonstrated. First, preincrement adds 1 to age and the result is output. Then, age’s current value is output and this variable is then decremented via postdecrement. What age values do you think are output?

The next two expression examples demonstrate string concatenation. First, "B" is concatenated to "A" and the resulting AB is output. Then, 32-bit integer value 5 is converted to a one-character string consisting of character 5, which is then concatenated to "A". The resulting A5 is output.

At this point, overflow is demonstrated. First, a 16-bit short integer variable named x is declared and initialized to the largest positive short integer: 32767. The preincrement operator is then applied to x and the result (-32768) is output.

Compile Listing 2-5, as follows:

```
javac CompoundExpressions.java
```

Assuming successful compilation, execute the following command to run this application:

```
java CompoundExpressions
```

You should observe the following output:

```
97
66
66
AB
A5
-32768
```
CHAPTER 2: Learning Language Fundamentals

Array Index Operator

The array index operator ([ ]) accesses an array element by presenting the location of that element as an integer index. This operator is specified after an array variable’s name, such as ages[0].

Indexes are relative to 0, which implies that ages[0] accesses the first element, whereas ages[6] accesses the seventh element. The index must be greater than or equal to 0 and less than the length of the array; otherwise, the virtual machine throws ArrayIndexOutOfBoundsException (consult Chapter 5 to learn about exceptions).

An array’s length is returned by appending “.length” to the array variable. For example, ages.length returns the length of (the number of elements in) the array that ages references. Similarly, matrix.length returns the number of row elements in the matrix two-dimensional array, whereas matrix[0].length returns the number of column elements assigned to the first row element of this array. (A two-dimensional array is essentially a one-dimensional row array of one-dimensional column arrays.)

Listing 2-6 presents a CompoundExpressions application that lets you start experimenting with the array index operator.

Listing 2-6. Experimenting with the Array Index Operator

```java
public class CompoundExpressions {
    public static void main(String[] args) {
        int[] ages = { 52, 28, 93, 16 };
        char gradeLetters[] = { 'A', 'B', 'C', 'D', 'F' };
        float[][] matrix = { { 1.0F, 2.0F, 3.0F }, { 4.0F, 5.0F, 6.0F } };
        System.out.println(ages[0]);
        System.out.println(gradeLetters[2]);
        System.out.println(matrix[1][2]);
        System.out.println(ages['']);
        ages[1] = 19;
        System.out.println(ages[1]);
    }
}
```

Listing 2-6’s main() method first declares and assigns arrays to variables ages, gradeLetters, and matrix. It then uses the array index operator to access the first element in the ages array (ages[0]), the third element in the gradeLetters array (gradeLetters[2]), and the third column element in the second row element of the matrix table array (matrix[1][2]).

Array indexes must be integer values. These values can be of byte integer, short integer, or integer type. However, they cannot be of long integer type because that could result in a loss of precision. The maximum number of elements that can be stored in an array is a bit less than the largest positive 32-bit integer; a long integer can be much larger than this value.

main() next demonstrates that you can also specify a character as an index value (ages['']). This is legal because Java supports a character-to-integer widening rule; it converts a character value to an integer value, which is then used as an index into the array (ages[2]). However, you should avoid using characters as array indexes because they’re not intuitive and are potentially error prone. For example, what element is accessed by ages['A']? The answer is the 66th element;
A's Unicode value is 65 and the first array index is 0. Given the previous four-element ages array, ages['A'] would result in ArrayIndexOutOfBoundsException.

Finally, main() demonstrates that you can also use the array index operator to assign a value to an array element. In this case, integer 19 is stored in the second array element, which is subsequently accessed and output.

Compile Listing 2-6 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

52
C
6.0
93
19

Assignment Operators

The assignment operator (=) assigns an expression’s result to a variable (as in int x = 4;). The types of the variable and expression must agree; otherwise, the compiler reports an error.

Java also supports several compound assignment operators that perform a specific operation and assign its result to a variable. For example, in pennies += 50;, the += operator evaluates the numeric expression on its right (50) and adds the result to the contents of the variable on its left (pennies). The other compound assignment operators behave in a similar way.

Bitwise Operators

The bitwise operators consist of bitwise AND (&), bitwise complement (~), bitwise exclusive OR (^), and bitwise inclusive OR (%). These operators are designed to work on the binary representations of their character or integral operands. Because this concept can be hard to understand if you haven’t previously worked with these operators in another language, check out Listing 2-7.

Listing 2-7. Experimenting with the Bitwise Operators

public class CompoundExpressions
{
    public static void main(String[] args)
    {
        System.out.println(~181);
        System.out.println(26 & 183);
        System.out.println(26 ^ 183);
        System.out.println(26 | 183);
    }
}
Compile Listing 2-7 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

```
-182
18
173
191
```

To make sense of these values, it helps to examine their 32-bit binary representations:

- 181 corresponds to 00000000000000000000000010110101
- 26 corresponds to 00000000000000000000000000011010
- 183 corresponds to 00000000000000000000000010110111

When you specify ~181, you end up flipping all of the bits: ~00000000000000000000000010110101 results in 11111111111111111111101001010. According to twos-complement representation, an integer whose leading bit is 1 is regarded to be negative, which is why ~181 equates to -182.

The expression 26 & 183 can be represented in binary as follows:

```
00000000000000000000000000011010
&
00000000000000000000000010110111
--------------------------------
00000000000000000000000000010010
```

The resulting binary value equates to 18.

The expression 26 ^ 183 can be represented in binary as follows:

```
00000000000000000000000000011010
^\n00000000000000000000000010110111
--------------------------------
00000000000000000000000010101101
```

The resulting binary value equates to 173.

The expression 26 | 183 can be represented in binary as follows:

```
00000000000000000000000000011010
|\n00000000000000000000000010110111
--------------------------------
00000000000000000000000010111111
```

The resulting binary value equates to 191.
CHAPTER 2: Learning Language Fundamentals

Cast Operator

The cast operator—(type)—attempts to convert the type of its operand to type. This operator exists because the compiler will not allow you to convert a value from one type to another in which information will be lost without specifying your intention to do so (via the cast operator). For example, when presented with short s = 1.65 + 3;, the compiler reports an error because attempting to convert a 64-bit double precision floating-point value to a 16-bit signed short integer results in the loss of the fraction .65, so s would contain 4 instead of 4.65.

Recognizing that information loss might not always be a problem, Java permits you to state your intention explicitly by casting to the target type. For example, short s = (short) 1.65 + 3; tells the compiler that you want 1.65 + 3 to be converted to a short integer, and that you realize that the fraction will disappear.

The following example provides another demonstration of the need for a cast operator:

```java
char c = 'A';
byte b = c;
```

The compiler reports an error about loss of precision when it encounters byte b = c;. The reason is that c can represent any unsigned integer value from 0 through 65535, whereas b can only represent a signed integer value from -128 through +127. Even though 'A' equates to +65, which can fit within b's range, c could just have easily been initialized to '\u0323', which wouldn’t fit.

The solution to this problem is to introduce a (byte) cast operator as follows, which causes the compiler to generate code to cast c’s character type to byte integer:

```java
byte b = (byte) c;
```

Java supports the following primitive-type conversions via cast operators:

- Byte integer to character
- Short integer to byte integer or character
- Character to byte integer or short integer
- Integer to byte integer, short integer, or character
- Long integer to byte integer, short integer, character, or integer
- Floating-point to byte integer, short integer, character, integer, or long integer
- Double precision floating-point to byte integer, short integer, character, integer, long integer, or floating-point

A cast operator isn’t always required when converting from more to fewer bits and where no data loss occurs. For example, when it encounters byte b = 100;, the compiler generates code that assigns integer 100 to byte integer variable b because 100 can easily fit into the 8-bit storage location assigned to this variable.

Listing 2-8 presents a CompoundExpressions application that lets you start experimenting with the cast operator.
Listing 2-8. Experimenting with the Cast Operator

public class CompoundExpressions
{
    public static void main(String[] args)
    {
        short s = (short) 1.65 + 3;
        System.out.println(s);

        char c = 'A';
        byte b = (byte) c;
        System.out.println(b);

        b = 100;
        System.out.println(b);

        s = 'A';
        System.out.println(s);

        s = (short) '\uac00';
        System.out.println(s);
    }
}

Listing 2-8's main() method first uses the (short) cast operator to narrow the double precision floating-point expression 1.65 + 3 to a 16-bit short integer that's ultimately assigned to short integer variable s. After outputting s's value (4), this method demonstrates the mandatory (byte) cast operator when converting from a 16-bit unsigned character type to an 8-bit signed byte integer type.

As previously mentioned, the (byte) cast operator isn't always required. For example, when assigning a 32-bit signed integer in the range of -128 through +127 to a byte integer variable, (byte) can be omitted because no information will be lost. Assigning 100 to b demonstrates this scenario.

In a similar way, various 16-bit unsigned character values (such as 'A') can be assigned to a 16-bit signed short integer variable without loss of information, and so the (short) cast operator can be avoided. However, other 16-bit unsigned character values don't fit into this range and must be cast to a short integer before assignment ('\uac00', for example).

Compile Listing 2-8 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

4
65
100
65
-21504
Conditional Operators

The conditional operators consist of conditional AND (&&), conditional OR (||), and conditional (?:).

The first two operators always evaluate their left operand (a Boolean expression that evaluates to true or false) and conditionally evaluate their right operand (another Boolean expression). The third operator evaluates one of two operands based on a third Boolean operand.

Conditional AND always evaluates its left operand and evaluates its right operand only when its left operand evaluates to true. For example, age > 64 && stillWorking first evaluates age > 64. If this subexpression is true, stillWorking is evaluated, and its true or false value (stillWorking is a Boolean variable) serves as the value of the overall expression. If age > 64 is false, stillWorking isn't evaluated.

Conditional OR always evaluates its left operand and evaluates its right operand only when its left operand evaluates to false. For example, value < 20 || value > 40 first evaluates value < 20. If this subexpression is false, value > 40 is evaluated, and its true or false value serves as the overall expression's value. If value < 20 is true, value > 40 isn't evaluated.

Conditional AND and conditional OR boost performance by preventing the unnecessary evaluation of subexpressions, which is known as short-circuiting. For example, if its left operand is false, there is no way that conditional AND's right operand can change the fact that the overall expression will evaluate to false.

If you aren’t careful, short-circuiting can prevent side effects (the results of subexpressions that persist after the subexpressions have been evaluated) from executing. For example, age > 64 && ++numEmployees > 5 increments numEmployees for only those employees whose ages are greater than 64. Incrementing numEmployees is an example of a side effect because the value in numEmployees persists after the subexpression ++numEmployees > 5 has evaluated.

The conditional operator is useful for making a decision by evaluating and returning one of two operands based upon the value of a third operand. The following example converts a Boolean value to its integer equivalent (1 for true and 0 for false):

```java
boolean b = true;
int i = b ? 1 : 0; // 1 assigns to i
```

Listing 2-9 presents a CompoundExpressions application that lets you start experimenting with the conditional operators.

Listing 2-9. Experimenting with the Conditional Operators

```java
public class CompoundExpressions
{
    public static void main(String[] args)
    {
        int age = 65;
        boolean stillWorking = true;
        System.out.println(age > 64 && stillWorking);
        age--;
        System.out.println(age > 64 && stillWorking);
        int value = 30;
        System.out.println(value < 20 || value > 40);
```
value = 10;
System.out.println(value < 20 || value > 40);
int numEmployees = 6;
age = 65;
System.out.println(age > 64 && ++numEmployees > 5);
System.out.println("numEmployees = " + numEmployees);
age = 63;
System.out.println(age > 64 && ++numEmployees > 5);
System.out.println("numEmployees = " + numEmployees);
boolean b = true;
int i = b ? 1 : 0; // 1 assigns to i
System.out.println("i = " + i);
b = false;
i = b ? 1 : 0; // 0 assigns to i
System.out.println("i = " + i);
}
}

Compile Listing 2-9 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

true
false
false
true
true
numEmployees = 7
false
numEmployees = 7
i = 1
i = 0

Equality Operators

The equality operators consist of equality (==) and inequality (!=). These operators compare their operands to determine whether they are equal or unequal. The former operator returns true when equal and the latter operator returns true when unequal. For example, each of 2 == 2 and 2 != 3 evaluates to true, whereas each of 2 == 4 and 4 != 4 evaluates to false.

You have to be careful when comparing floating-point expressions for equality. For example, what does System.out.println(0.3 == 0.1 + 0.1 + 0.1); output? If you guessed that the output is true, you would be wrong. Instead, the output is false.

The reason for this nonintuitive output is that 0.1 cannot be represented exactly in memory. The error compounds when this value is added to itself. For example, if you executed System.out.println(0.1 + 0.1 + 0.1);, you would observe 0.30000000000000004, which doesn't equal 0.3.

When it comes to object operands (I discuss objects in Chapter 3), these operators don't compare their contents. Instead, object references are compared. For example, "abc" == "xyz" doesn't compare a with x. Because string literals are really String objects (Chapter 7 discusses the String class), == compares the references to these objects.
Logical Operators

The logical operators consist of logical AND (\&), logical complement (!), logical exclusive OR (^), and logical inclusive OR (|). Although these operators are similar to their bitwise counterparts, whose operands must be integer/character, the operands passed to the logical operators must be Boolean. For example, !false returns true. Also, when confronted with age > 64 & stillWorking, logical AND evaluates both subexpressions; there's no short-circuiting. This same pattern holds for logical exclusive OR and logical inclusive OR.

Listing 2-10 presents a CompoundExpressions application that lets you start experimenting with the logical operators.

Listing 2-10. Experimenting with the Logical Operators

```java
public class CompoundExpressions {
    public static void main(String[] args) {
        System.out.println(!false);
        int age = 65;
        boolean stillWorking = true;
        System.out.println(age > 64 & stillWorking);
        System.out.println();
        boolean result = true & true;
        System.out.println("true & true: " + result);
        result = true & false;
        System.out.println("true & false: " + result);
        result = false & true;
        System.out.println("false & true: " + result);
        result = false & false;
        System.out.println("false & false: " + result);
        System.out.println();
        result = true | true;
        System.out.println("true | true: " + result);
        result = true | false;
        System.out.println("true | false: " + result);
        result = false | true;
        System.out.println("false | true: " + result);
        result = false | false;
        System.out.println("false | false: " + result);
        System.out.println();
        result = true ^ true;
        System.out.println("true ^ true: " + result);
        result = true ^ false;
        System.out.println("true ^ false: " + result);
        result = false ^ true;
        System.out.println("false ^ true: " + result);
        result = false ^ false;
        System.out.println("false ^ false: " + result);
    }
}```
After outputting the results of the `!false` and `age > 64 & stillWorking` expressions, main() outputs three truth tables that show how logical AND, logical inclusive OR, and logical exclusive OR behave when their operands are true or false. It then demonstrates that short-circuiting is ignored by incrementing `numEmployees` when `age > 64` returns true.

Compile Listing 2-10 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

```
true
true

true & true: true
true & false: false
false & true: false
false & false: false

true | true: true
true | false: true
false | true: true
false | false: false

true ^ true: false
true ^ false: true
false ^ true: true
false ^ false: false

false
2
```

**Member Access Operator**

The member access operator (.) is used to access a class's members or an object's members. For example, `String s = "Hello"; int len = s.length();` returns the length of the string assigned to variable `s`. It does so by calling the `length()` method member of the `String` class. In Chapter 3, I discuss member access in more detail.

Arrays are special objects that have a single `length` member. When you specify an array variable followed by the member access operator, followed by `length`, the resulting expression returns the number of elements in the array as a 32-bit integer. For example, `ages.length` returns the length of (the number of elements in) the array that `ages` references.
Method Call Operator

The method call operator (\(\)\) is used to signify that a method (discussed in Chapter 3) is being called. Also, it identifies the number, order, and types of arguments that are passed to the method to be picked up by the method’s parameters. For example, in the `System.out.println("Hello");` method call, (\(\)\) signifies that a method named `println` is being called with one argument: "Hello".

Multiplicative Operators

The multiplicative operators consist of multiplication (\(*\)\), division (\(/\)\), and remainder (\(\%\)\). Multiplication returns the product of its operands (such as \(6 * 4\) returns 24), division returns the quotient of dividing its left operand by its right operand (such as \(6 / 4\) returns 1), and remainder returns the remainder of dividing its left operand by its right operand (such as \(6 \% 4\) returns 2).

The multiplication, division, and remainder operators can yield values that overflow or underflow the limits of the resulting value’s type. For example, multiplying two large positive 32-bit integer values can produce a value that cannot be represented as a 32-bit integer value. The result is said to overflow. Java doesn’t detect overflows and underflows.

Dividing a numeric value by 0 (via the division or remainder operator) also results in interesting behavior. Dividing an integer value by integer 0 causes the operator to throw an `ArithmeticException` object (Chapter 5 covers exceptions). Dividing a floating-point/double precision floating-point value by 0 causes the operator to return +infinity or -infinity, depending on whether the dividend is positive or negative. Finally, dividing floating-point 0 by 0 causes the operator to return NaN (Not a Number).

Listing 2-11 presents a `CompoundExpressions` application that lets you start experimenting with the multiplicative operators.

**Listing 2-11. Experimenting with the Multiplicative Operators**

```java
public class CompoundExpressions
{
    public static void main(String[] args)
    {
        short age = 65;
        System.out.println(age * 1000);
        System.out.println(1.0 / 0.0);
        System.out.println(10 % 4);
        System.out.println(3 / 0);
    }
}
```
Compile Listing 2-11 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

65000
Infinity
2
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at CompoundExpressions.main(CompoundExpressions.java:9)

Object Creation Operator

The object creation operator (new) creates an object from a class, and it also creates an array from an initializer. These topics are discussed in Chapter 3.

Relational Operators

The relational operators consist of greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), and type checking (instanceof). The former four operators compare their operands and return true when the left operand is (respectively) greater than, greater than or equal to, less than, or less than or equal to the right operand. For example, each of 5.0 > 3, 2 >= 2, 16.1 < 303.3, and 54.0 <= 54.0 evaluates to true.

The type-checking operator is used to determine if an object belongs to a specific type, returning true when this is the case. For example, "abc" instanceof String returns true because "abc" is a String object. I discuss this operator more fully in Chapter 4.

Shift Operators

The shift operators consist of left shift (<<), signed right shift (>>), and unsigned right shift (>>>). Left shift shifts the binary representation of its left operand leftward by the number of positions specified by its right operand. Each shift is equivalent to multiplying by 2. For example, 2 << 3 shifts 2's binary representation left by three positions; the result is equivalent to multiplying 2 by 8.

Each of signed and unsigned right shift shifts the binary representation of its left operand rightward by the number of positions specified by its right operand. Each shift is equivalent to dividing by 2. For example, 16 >> 3 shifts 16's binary representation right by three positions; the result is equivalent to dividing 16 by 8.

The difference between signed and unsigned right shift is what happens to the sign bit during the shift. Signed right shift includes the sign bit in the shift, whereas unsigned right shift ignores the sign bit. As a result, signed right shift preserves negative numbers, but unsigned right shift doesn't. For example, -4 >> 1 (the equivalent of -4 / 2) evaluates to -2, whereas -4 >>> 1 evaluates to 2147483646.

Listing 2-12 presents a CompoundExpressions application that lets you start experimenting with the shift operators.
Listing 2-12. Experimenting with the Shift Operators

```java
public class CompoundExpressions {
    public static void main(String[] args) {
        System.out.println(2 << 3);
        System.out.println(16 >> 3);
        System.out.println(-4 >> 1);
        System.out.println(-4 >>> 1);
    }
}
```

Compile Listing 2-12 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

16
2
-2
2147483646

**Tip** The shift operators are faster than multiplying or dividing by powers of 2.

**Unary Minus/Plus Operators**

Unary minus (-) and unary plus (+) are the simplest of all operators. Unary minus returns the negative of its operand (such as -5 returns -5 and --5 returns 5), whereas unary plus returns its operand verbatim (such as +5 returns 5 and ++-5 returns -5). Unary plus is not commonly used, but it is presented for completeness.

**Precedence and Associativity**

When evaluating a compound expression, Java takes each operator's precedence (level of importance) into account to ensure that the expression evaluates as expected. For example, when presented with the expression 60 + 3 * 6, you expect multiplication to be performed before addition (multiplication has higher precedence than addition) and the final result to be 78. You don’t expect addition to occur first, yielding a result of 378.

**Note** Table 2-3’s rightmost column presents a value that indicates an operator’s precedence: the higher the number, the higher the precedence. For example, addition’s precedence level is 10 and multiplication’s precedence level is 11, which means that multiplication is performed before addition.
Precedence can be circumvented by introducing open and close parentheses, ( and ), into the expression, where the innermost pair of nested parentheses is evaluated first. For example, evaluating $2 \times ((60 + 3) \times 6)$ results in $(60 + 3)$ being evaluated first, $(60 + 3) \times 6$ being evaluated next, and the overall expression being evaluated last. Similarly, in the expression $60 / (3 - 6)$, subtraction is performed before division.

During evaluation, operators with the same precedence level (such as addition and subtraction, which both have level 10) are processed according to their associativity (a property that determines how operators having the same precedence are grouped when parentheses are missing).

For example, expression $9 \times 4 / 3$ is evaluated as if it was $(9 \times 4) / 3$ because $\times$ and $/$ are left-to-right associative operators. In contrast, expression $x = y = z = 100$ is evaluated as if it was $x = (y = (z = 100))$, where 100 is assigned to $z$, $z$'s new value (100) is assigned to $y$, and $y$'s new value (100) is assigned to $x$ because $=$ is a right-to-left associative operator.

Most of Java's operators are left-to-right associative. Right-to-left associative operators include assignment, bitwise complement, cast, compound assignment, conditional, logical complement, object creation, predecrement, preincrement, unary minus, and unary plus.

Listing 2-13 presents a CompoundExpressions application that lets you start experimenting with precedence and associativity.

**Listing 2-13. Experimenting with Precedence and Associativity**

```java
class CompoundExpressions {
    public static void main(String[] args) {
        System.out.println(60 + 3 * 6);
        System.out.println(2 * ((60 + 3) * 6));
        System.out.println(9 * 4 / 3);
        int x, y, z;
        x = y = z = 100;
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);
        int i = 0x12345678;
        byte b = (byte) (i & 255);
        System.out.println(b);
        System.out.println("b == 0x78: " + (b == 0x78));
        b = (byte) ((i >> 8) & 255);
        System.out.println(b);
        System.out.println("b == 0x56: " + (b == 0x56));
        b = (byte) ((i >> 16) & 255);
        System.out.println(b);
        System.out.println("b == 0x34: " + (b == 0x34));
        b = (byte) ((i >> 24) & 255);
        System.out.println(b);
        System.out.println("b == 0x12: " + (b == 0x12));
    }
}
```
You'll often find yourself needing to use open and close parentheses to change an expression’s evaluation order. For example, consider the second part of the `main()` method, which extracts each of the 4 bytes in the 32-bit value assigned to integer variable `i` and outputs this byte.

After processing the declaration and initialization of 32-bit integer variable `i` `(int i = 0x12345678;`), the compiler encounters `byte b = (byte) (i & 255);`. It generates bytecode that first evaluates expression `i & 255`, which returns a 32-bit result, passes this result to the `(byte)` cast operator to convert it to an 8-bit result, and assigns the 8-bit result to 8-bit byte integer variable `b`.

Suppose the parentheses were absent, resulting in `byte b = (byte) i & 255;`. The compiler would then report an error about loss of precision because it interprets this expression as follows.

1. Cast variable `i` to an 8-bit byte integer. The cast operator is a unary operator that takes only one operand. Also, cast has higher precedence (12) than bitwise AND (6) so cast is evaluated first.
2. Widen `i` to a 32-bit integer as the left operand of bitwise AND (`&`). Operand 255 is already a 32-bit integer.
3. Apply bitwise AND to these operands. The result is a 32-bit integer.
4. Attempt to assign the 32-bit integer result to 8-bit byte integer variable `b`.

The 32-bit integer result can vary from 0 through 255. However, the largest positive integer that `b` can store is 127. If the result ranges from 128 through 255, it will be converted to -1 through -128. This is a loss of precision and so the compiler reports an error.

Another example where `main()` uses open and close parentheses to change evaluation order is `System.out.println("b == 0x78: " + (b == 0x78));`. When the parentheses are missing, the compiler reports an “incomparable types: String and int” error. It does so because string concatenation has higher precedence (10) than equality (7). As a result, the compiler interprets the expression (without parentheses) as follows.

1. Convert `b`’s value to a string.
2. Concatenate this string to "b == 0x78: ".
3. Compare the resulting string with 32-bit integer `0x78` for equality, which is illegal.
Compile Listing 2-13 (javac CompoundExpressions.java) and run this application (java CompoundExpressions). You should observe the following output:

78
756
12
100
100
100
120
b == 0x78: true
86
b == 0x56: true
52
b == 0x34: true
18
b == 0x12: true

Note  Unlike languages such as C++, Java doesn’t let you overload operators. However, Java overloads the + (addition and string concatenation), ++ (preincrement and postincrement), and -- (predecrement and postdecrement) operator symbols.

Learning Statements

Statements are the workhorses of a program. They assign values to variables, control a program’s flow by making decisions and/or repeatedly executing other statements, and perform other tasks. A statement can be expressed as a simple statement or as a compound statement.

- A simple statement is a single standalone source code instruction for performing some task; it’s terminated with a semicolon.
- A compound statement is a (possibly empty) sequence of simple and other compound statements sandwiched between open and close brace delimiters; a delimiter is a character that marks the beginning or end of some section. A method body (such as the main() method’s body) is an example. Compound statements can appear wherever simple statements appear and are alternatively referred to as blocks.

In this section I introduce you to many of Java’s statements. Additional statements are covered in later chapters. For example, in Chapter 3 I discuss the return statement.
Assignment Statements

The *assignment statement* assigns a value to a variable. This statement begins with a variable name, continues with the assignment operator (=) or a compound assignment operator (such as +=), and concludes with an assignment-compatible expression and a semicolon. The following are three examples:

```
x = 10;
ages[0] = 25;
counter += 10;
```

The first example assigns integer 10 to variable `x`, which is presumably of type integer as well. The second example assigns integer 25 to the first element of the `ages` array. The third example adds 10 to the value stored in `counter` and stores the sum in `counter`.

Note: Initializing a variable in the variable’s declaration (such as `int counter = 1;`) can be thought of as a special form of the assignment statement.

Decision Statements

The previously described conditional operator (?:) is useful for choosing between two expressions to evaluate and cannot be used to choose between two statements. For this purpose, Java supplies three decision statements: if, if-else, and switch.

If Statement

The *if statement* evaluates a Boolean expression and executes another statement if this expression evaluates to true. It has the following syntax:

```
if (Boolean expression)
    statement
```

This statement consists of reserved word if, followed by a *Boolean expression* in parentheses, followed by a *statement* to execute when *Boolean expression* evaluates to true.

The following example demonstrates the if statement:

```
if (numMonthlySales > 100)
    wage += bonus;
```

If the number of monthly sales exceeds 100, `numMonthlySales > 100` evaluates to true and the `wage += bonus;` assignment statement executes. Otherwise, this assignment statement doesn’t execute.
CHAPTER 2: Learning Language Fundamentals

If-Else Statement

The *if-else statement* evaluates a Boolean expression and executes one of two statements depending on whether this expression evaluates to true or false. It has the following syntax:

```
if (Boolean expression)
  statement1
else
  statement2
```

This statement consists of reserved word *if*, followed by a *Boolean expression* in parentheses, followed by a *statement1* to execute when *Boolean expression* evaluates to true, followed by a *statement2* to execute when *Boolean expression* evaluates to false.

The following example demonstrates the if-else statement:

```
if ((n & 1) == 1)
  System.out.println("odd");
else
  System.out.println("even");
```

This example assumes the existence of an int variable named n that’s been initialized to an integer. It then proceeds to determine if the integer is odd (not divisible by 2) or even (divisible by 2).

The Boolean expression first evaluates n & 1, which bitwise ANDs n’s value with 1. It then compares the result to 1. If they’re equal, a message stating that n’s value is odd outputs; otherwise, a message stating that n’s value is even outputs.

The parentheses are required because == has higher precedence than &. Without these parentheses, the expression’s evaluation order would change to first evaluating 1 == 1 and then trying to bitwise AND the Boolean result with n’s integer value. This order results in a compiler error message because of a type mismatch: you cannot bitwise AND an integer with a Boolean value.

You could rewrite this if-else statement example to use the conditional operator, as follows:

```
System.out.println((n & 1) == 1 ? "odd" : "even");
```
However, you cannot do so with the following example:

```java
if ((n & 1) == 1)
    odd();
else
    even();
```

This example assumes the existence of `odd()` and `even()` methods that don’t return anything. Because the conditional operator requires that each of its second and third operands evaluates to a value, the compiler reports an error when attempting to compile `(n & 1) == 1 ? odd() : even()`.

You can chain multiple if-else statements together, resulting in the following syntax:

```java
if (Boolean expression1)
    statement1
else
    if (Boolean expression2)
        statement2
    else
    ...
else
    statementN
```

If `Boolean expression1` evaluates to true, `statement1` executes. Otherwise, if `Boolean expression2` evaluates to true, `statement2` executes. This pattern continues until one of these expressions evaluates to true and its corresponding statement executes, or the final `else` is reached and `statementN` (the default statement) executes.

Listing 2-14 presents a `GradeLetters` application that demonstrates chaining together multiple if-else statements.

**Listing 2-14. Experimenting with If-Else Chaining**

```java
public class GradeLetters {
    public static void main(String[] args) {
        int testMark = 69;
        char gradeLetter;

        if (testMark >= 90) {
            gradeLetter = 'A';
            System.out.println("You aced the test.");
        }
        else if (testMark >= 80) {
            gradeLetter = 'B';
            System.out.println("You did very well on this test.");
        }
    }
}
```
else
    if (testMark >= 70)
    {
        gradeLetter = 'C';
        System.out.println("You'll need to study more for future tests.");
    }
    else
    if (testMark >= 60)
    {
        gradeLetter = 'D';
        System.out.println("Your test result suggests that you need a tutor.");
    }
    else
    {
        gradeLetter = 'F';
        System.out.println("Your fail and need to attend summer school.");
    }

    System.out.println("Your grade is " + gradeLetter + ".");
}

Compile Listing 2-14 as follows:

javac GradeLetters.java

Execute the resulting application as follows:

java GradeLetters

You should observe the following output:

Your test result suggests that you need a tutor.
Your grade is D.

---

**Dangling-Else Problem**

When if and if-else are used together and the source code isn't properly indented, it can be difficult to determine which if associates with the else. See the following, for example:

```java
if (car.door.isOpen())
    if (car.key.isPresent())
        car.start();
else car.door.open();
```
Did the developer intend for the else to match the inner if, but improperly formatted the code to make it appear otherwise? This reformatted possibility appears below:

```java
if (car.door.isOpen())
  if (car.key.isPresent())
    car.start();
  else
    car.door.open();
```

If `car.door.isOpen()` and `car.key.isPresent()` each return true, `car.start()` executes. If `car.door.isOpen()` returns true and `car.key.isPresent()` returns false, `car.door.open();` executes. Attempting to open an open door makes no sense.

The developer must have wanted the else to match the outer if but forgot that else matches the nearest if. This problem can be fixed by surrounding the inner if with braces, as follows:

```java
if (car.door.isOpen())
{
  if (car.key.isPresent())
    car.start();
}
else
  car.door.open();
```

When `car.door.isOpen()` returns true, the compound statement executes. When this method returns false, `car.door.open();` executes, which makes sense.

Forgetting that else matches the nearest if and using poor indentation to obscure this fact is known as the *dangling-else problem*.

---

**Switch Statement**

The *switch statement* lets you choose from among several execution paths in a more efficient manner than with equivalent chained if-else statements. It has the following syntax:

```java
switch (selector expression)
{
  case value1: statement1 [break;]
  case value2: statement2 [break;]
  ...
  case valueN: statementN [break;]
  [default: statement]
}
```
This statement consists of reserved word `switch`, followed by a `selector expression` in parentheses, followed by a body of cases. The `selector expression` is any expression that evaluates to an integer or character value. For example, it might evaluate to a 32-bit integer or to a 16-bit character.

Each case begins with reserved word `case`, continues with a literal value and a colon character (:) , continues with a statement to execute; and optionally concludes with a break statement, which causes execution to continue after the switch statement.

After evaluating the `selector expression`, `switch` compares this value with each case’s value until it finds a match. When there is a match, the case’s statement is executed. For example, when the `selector expression’s value matches value1, statement1 executes.

The optional break statement (anything placed in square brackets is optional), which consists of reserved word `break` followed by a semicolon, prevents the flow of execution from continuing with the next case’s statement. Instead, execution continues with the first statement following switch.

```
Note You’ll usually place a break statement after a case’s statement. Forgetting to include break can lead to a hard-to-find bug. However, there are situations where you want to group several cases together and have them execute common code. In this situation, you would omit the break statement from the participating cases.
```

If none of the cases’ values match the `selector expression’s value, and if a default case (signified by the `default` reserved word followed by a colon) is present, the default case’s statement is executed.

The following example demonstrates this statement:

```
switch (direction)
{
    case  0: System.out.println("You are travelling north."); break;
    case  1: System.out.println("You are travelling east."); break;
    case  2: System.out.println("You are travelling south."); break;
    case  3: System.out.println("You are travelling west."); break;
    default: System.out.println("You are lost.");
}
```

This example assumes that `direction` stores an integer value. When this value is in the range 0-3, an appropriate direction message is output; otherwise, a message about being lost is output.

```
Note This example hardcodes values 0, 1, 2, and 3, which isn’t a good idea in practice. Instead, constants should be used. Chapter 3 introduces you to constants.
```

### Loop Statements

It’s often necessary to repeatedly execute a statement; this repeated execution is called a loop. Java provides three kinds of loop statements: for, while, and do-while. In this section, I first discuss these statements. I then examine the topic of looping over the empty statement. Finally, I discuss the break, labeled break, continue, and labeled continue statements for prematurely ending all or part of a loop.

www.it-ebooks.info
For Statement

The for statement lets you loop over a statement a specific number of times or even indefinitely. It has the following syntax:

```
for ([initialize]; [test]; [update])
    statement
```

This statement consists of reserved word for, followed by a header in parentheses, followed by a statement to execute. The header consists of an optional initialize section, followed by an optional test section, followed by an optional update section. A nonoptional semicolon separates each of the first two sections from the next section.

The initialize section consists of a comma-separated list of variable declarations or variable assignments. Some or all of these variables are typically used to control the loop’s duration and are known as loop-control variables.

The test section consists of a Boolean expression that determines how long the loop executes. Execution continues as long as this expression evaluates to true.

Finally, the update section consists of a comma-separated list of expressions that typically modify the loop-control variables.

The for statement is perfect for iterating (looping) over an array. Each iteration (loop execution) accesses one of the array’s elements via an array[index] expression, where array is the array whose element is being accessed and index is the zero-based location of the element being accessed.

The following example uses for to iterate over the array of command-line arguments passed to main():

```java
public static void main(String[] args)
{
    for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);
}
```

The initialization section declares variable i for controlling the loop, the test section compares i’s current value to the length of the args array to ensure that this value is less than the array’s length, and the update section increments i by 1. The for-based loop continues until i’s value equals the array’s length.

Each array element is accessed via the args[i] expression, which returns this array’s ith element’s value (which happens to be a String object in this example). The first value is stored in args[0].

Note Although I’ve named the array containing command-line arguments args, this name isn’t mandatory. I could as easily have named it arguments (or even some_other_name).
Listing 2-15 presents a DumpMatrix application that uses a for-based loop to output the contents of a two-dimensional matrix array.

Listing 2-15. Iterating over a Two-Dimensional Array's Rows and Columns

```java
public class DumpMatrix {
    public static void main(String[] args) {
        float[][] matrix = { { 1.0F, 2.0F, 3.0F }, { 4.0F, 5.0F, 6.0F }};
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++)
                System.out.print(matrix[row][col] + " ");
            System.out.print("\n");
        }
    }
}
```

expression `matrix.length` returns the number of rows in this tabular array. For each row, expression `matrix[row].length` returns the number of columns for that row. This latter expression suggests that each row can have a different number of columns, although each row has the same number of columns in the example.

`System.out.print()` is closely related to `System.out.println()`. Unlike the latter method, `System.out.print()` outputs its argument without a trailing newline.

Compile Listing 2-15 as follows:

```
javac DumpMatrix.java
```

Execute the resulting application as follows:

```
java DumpMatrix
```

You should observe the following output:

```
1.0 2.0 3.0
4.0 5.0 6.0
```

### While Statement

The *while statement* repeatedly executes another statement while its Boolean expression evaluates to true. It has the following syntax:

```
while (Boolean expression)  
    statement
```

This statement consists of reserved word *while*, followed by a parenthesized *Boolean expression*, followed by a *statement* to execute repeatedly. The *while statement* first evaluates the *Boolean expression*.
CHAPTER 2: Learning Language Fundamentals

expression. If it’s true, while executes the other statement. Once again, the Boolean expression is evaluated. If it’s still true, while re-executes the statement. This cyclic pattern continues.

Prompting the user to enter a specific character is one situation where while is useful. For example, suppose that you want to prompt the user to enter a specific uppercase letter or its lowercase equivalent. The following example provides a demonstration:

```java
int ch = 0;
while (ch != 'C' && ch != 'c')
{
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
}
```

This example first initializes variable ch. This variable must be initialized; otherwise, the compiler will report an uninitialized variable when it tries to read ch’s value in the while statement’s Boolean expression.

This expression uses the conditional AND operator (&&) to test ch’s value. This operator first evaluates its left operand, which happens to be expression ch != 'C'. (The != operator converts 'C' from 16-bit unsigned char type to 32-bit signed int type before the comparison.)

If ch doesn’t contain C (it doesn’t at this point; 0 was just assigned to ch), this expression evaluates to true.

The && operator next evaluates its right operand, which happens to be expression ch != 'c'. Because this expression also evaluates to true, conditional AND returns true and while executes the compound statement.

The compound statement first outputs, via the System.out.println() method call, a message that prompts the user to press the C key with or without the Shift key. It next reads the entered keystroke via System.in.read(), saving its integer value in ch.

From left to write, System identifies a standard class of system utilities, in identifies an object located in System that provides methods for inputting one or more bytes from the standard input device, and read() returns the next byte (or -1 when there are no more bytes).

After this assignment, the compound statement ends and while re-evaluates its Boolean expression.

Suppose ch contains C’s integer value. Conditional AND evaluates ch != 'C', which evaluates to false. Detecting that the expression is already false, conditional AND short-circuits its evaluation by not evaluating its right operand and returns false. The while statement subsequently detects this value and terminates.

Suppose ch contains c’s integer value. Conditional AND evaluates ch != 'C', which evaluates to true. Detecting that the expression is true, conditional AND evaluates ch != 'c', which evaluates to false. Once again, the while statement terminates.
Note  A for statement can be coded as a while statement. For example,

```java
for (int i = 0; i < 10; i++)
    System.out.println(i);
```

is equivalent to

```java
int i = 0;
while (i < 10)
{
    System.out.println(i);
    i++;
}
```

---

**Do-While Statement**

The *do-while statement* repeatedly executes a statement while its Boolean expression evaluates to true. Unlike while, which evaluates the Boolean expression at the top of the loop, do-while evaluates the Boolean expression at the bottom of the loop. It has the following syntax:

```java
do
    statement
while (Boolean expression);
```

This statement consists of the `do` reserved word, followed by a `statement` to execute repeatedly, followed by the `while` reserved word, followed by a parenthesized `Boolean expression`, followed by a semicolon.

The do-while statement first executes the other `statement`. It then evaluates the `Boolean expression`. If it's true, do-while executes the other `statement`. Once again, the `Boolean expression` is evaluated. If it's still true, do-while re-executes the `statement`. This cyclic pattern continues.

The following example demonstrates do-while prompting the user to enter a specific uppercase letter or its lowercase equivalent:

```java
int ch;
do
{
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
}
while (ch != 'C' && ch != 'c');
```

This example is similar to its predecessor. Because the compound statement is no longer executed before the test, it’s no longer necessary to initialize `ch`—`ch` is assigned `System.in.read()`’s return value before the Boolean expression’s evaluation.
Looping Over the Empty Statement

Java refers to a semicolon character appearing by itself as the *empty statement*. It’s sometimes convenient for a loop statement to execute the empty statement repeatedly. The actual work performed by the loop statement takes place in the statement header.

Consider the following example:

```java
for (String line; (line = readLine()) != null; System.out.println(line));
```

This example uses `for` to present a programming idiom for copying lines of text that are read from some source, via the fictitious `readLine()` method in this example, to some destination, via `System.out.println()` in this example. Copying continues until `readLine()` returns null. Note the semicolon (empty statement) at the end of the line.

**Caution** Be careful with the empty statement because it can introduce subtle bugs into your code. For example, the following loop is supposed to output the string *Hello* on 10 lines. Instead, only one instance of this string is output, because it’s the empty statement and not `System.out.println()` that’s executed 10 times:

```java
for (int i = 0; i < 10; i++); // this ; represents the empty statement
    System.out.println("Hello");
```

Break and Labeled Break Statements

What do `for (;;);`, `while (true);` and `do;while (true);` have in common? Each of these loop statements presents an extreme example of an *infinite loop* (a loop that never ends). An infinite loop is something that you should avoid because its unending execution causes your application to hang, which isn’t desirable from the point of view of your application’s users.

**Caution** An infinite loop can also arise from a loop’s Boolean expression comparing a floating-point value with a nonzero value via the equality or inequality operator, because many floating-point values have inexact internal representations. For example, the following example never ends because `0.1` doesn’t have an exact internal representation:

```java
for (double d = 0.0; d != 1.0; d += 0.1)
    System.out.println(d);
```

However, there are times when it’s handy to code a loop as if it were infinite by using one of the aforementioned programming idioms. For example, you might code a `while (true)` loop that repeatedly prompts for a specific keystroke until the correct key is pressed. When the correct key is pressed, the loop must end. Java provides the break statement for this purpose.
The break statement transfers execution to the first statement following a switch statement (as discussed earlier) or a loop. In either scenario, this statement consists of reserved word break followed by a semicolon.

The following example uses break with an if decision statement to exit a while (true)-based infinite loop when the user presses the C or c key:

```java
int ch;
while (true)
{
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
    if (ch == 'C' || ch == 'c')
        break;
}
```

The break statement is also useful in the context of a finite loop. For example, consider a scenario where an array of values is searched for a specific value, and you want to exit the loop when this value is found. Listing 2-16 presents an EmployeeSearch application that demonstrates this scenario.

**Listing 2-16. Searching for a Specific Employee ID**

```java
public class EmployeeSearch
{
    public static void main(String[] args)
    {
        int[] employeeIDs = { 123, 854, 567, 912, 224 };   
        int employeeSearchID = 912;
        boolean found = false;
        for (int i = 0; i < employeeIDs.length; i++)
        {
            if (employeeSearchID == employeeIDs[i])
            {
                found = true;
                break;
            }
        }
        System.out.println((found) ? "employee " + employeeSearchID + " exists" : "no employee ID matches " + employeeSearchID);
    }
}
```

Listing 2-16 uses for and if statements to search an array of employee IDs to determine if a specific employee ID exists. If this ID is found, if’s compound statement assigns true to found. Because there’s no point in continuing the search, it then uses break to quit the loop.

Compile Listing 2-16 as follows:

```bash
javac EmployeeSearch.java
```

Run this application as follows:

```bash
java EmployeeSearch
```
You should observe the following output:

```
employee 912 exists
```

The *labeled break statement* transfers execution to the first statement following the loop that’s prefixed by a *label* (an identifier followed by a colon). It consists of reserved word break, followed by an identifier for which the matching label must exist. Furthermore, the label must immediately precede a loop statement.

The labeled break is useful for breaking out of *nested loops* (loops within loops). The following example reveals the labeled break statement transferring execution to the first statement that follows the outer for loop:

```
outer:
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        if (i == 1 && j == 1)
            break outer;
        else
            System.out.println("i=\(i\)\), j=\(j\)\);
System.out.println("Both loops terminated.");
```

When i’s value is 1 and j’s value is 1, break outer; is executed to terminate both for loops. This statement transfers execution to the first statement after the outer for loop, which happens to be System.out.println("Both loops terminated.");.

The following output is generated:

```
i=0, j=0
i=0, j=1
i=0, j=2
i=1, j=0
Both loops terminated.
```

**Continue and Labeled Continue Statements**

The *continue statement* skips the remainder of the current loop iteration, re-evaluates the loop’s Boolean expression, and performs another iteration (if true) or terminates the loop (if false). Continue consists of reserved word continue followed by a semicolon.

Consider a while loop that reads lines from a source and processes nonblank lines in some manner. Because it shouldn’t process blank lines, while skips the current iteration when a blank line is detected, as demonstrated in the following example:

```
String line;
while ((line = readLine()) != null)
{
    if (isBlank(line))
        continue;
    processLine(line);
}
```
This example employs a fictitious `isBlank()` method to determine if the currently read line is blank. If this method returns true, it executes the continue statement to skip the rest of the current iteration and read the next line whenever a blank line is detected. Otherwise, the fictitious `processLine()` method is called to process the line’s contents.

Look carefully at this example, and you should realize that the continue statement isn’t needed. Instead, this listing can be shortened via refactoring (rewriting source code to improve its readability, organization, or reusability), as demonstrated in the following example:

```java
String line;
while ((line = readLine()) != null)
{
    if (!isBlank(line))
        processLine(line);
}
```

This example’s refactoring modifies if’s Boolean expression to use the logical complement operator (!). Whenever `isBlank()` returns false, this operator flips this value to true and if executes `processLine()`. Although continue isn’t necessary in this example, you’ll find it convenient to use this statement in more complex code where refactoring isn’t as easy to perform.

The labeled continue statement skips the remaining iterations of one or more nested loops and transfers execution to the labeled loop. It consists of reserved word `continue`, followed by an identifier for which a matching label must exist. Furthermore, the label must immediately precede a loop statement.

Labeled continue is useful for breaking out of nested loops while still continuing to execute the labeled loop. The following example reveals the labeled continue statement terminating the inner for loop’s iterations:

```java
outer:
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        if (i == 1 && j == 1)
            continue outer;
        else
            System.out.println("i=" + i + ", j=" + j);
System.out.println("Both loops terminated.");
```

When i’s value is 1 and j’s value is 1, `continue outer;` is executed to terminate the inner for loop and continue with the outer for loop at its next value of i. Both loops continue until they finish.

The following output is generated:

```
i=0, j=0
i=0, j=1
i=0, j=2
i=1, j=0
i=2, j=0
i=2, j=1
i=2, j=2
Both loops terminated.
```
EXERCISES

The following exercises are designed to test your understanding of Chapter 2's content.

1. What is Unicode?

2. What is a comment?

3. Identify the three kinds of comments that Java supports.

4. What is an identifier?

5. True or false: Java is a case-insensitive language.

6. What is a type?

7. Define primitive type.

8. Identify all of Java's primitive types.

9. Define user-defined type.

10. Define array type.

11. What is a variable?

12. What is an expression?

13. Identify the two expression categories.

14. What is a literal?

15. Is string literal "The quick brown fox \jumps\ over the lazy dog." legal or illegal? Why?

16. What is an operator?

17. Identify the difference between a prefix operator and a postfix operator.

18. What is the purpose of the cast operator?

19. What is precedence?

20. True or false: Most of Java’s operators are left-to-right associative.

21. What is a statement?

22. What is the difference between the while and do-while statements?

23. What is the difference between the break and continue statements?

24. Write a Compass application (the class is named Compass) whose main() method encapsulates the direction-oriented switch statement example presented earlier in this chapter. You’ll need to declare a direction variable with the appropriate type and initialize this variable.
25. Create a Triangle application whose Triangle class’s main() method uses a pair of nested for statements along with System.out.print() to output a 10-row triangle of asterisks, where each row contains an odd number of asterisks (1, 3, 5, 7, and so on), as shown below:

*  
***  
*****  
*******  
********  
*********  
**********  
***********  
************  
*************  

Compile and run this application.

26. Write a pair of PromptForC applications whose main() methods encapsulate the while and do-while examples that prompt for input of letter c or letter C. Because each method uses System.in.read() to obtain input, append throws java.io.Exception to the main() method header.

Summary

Before developing Java applications, you need to understand the structure of a Java application. Essentially, every application drills down to a single class that declares a public static void main(String[] args) method.

Source code needs to be documented so that you (and any others who have to maintain it) can understand it, now and later. Java provides the comment feature for embedding documentation in source code. Single-line, multiline, and documentation comments are supported.

A single-line comment occupies all or part of a single line of source code. This comment begins with the // character sequence and continues with explanatory text. The compiler ignores everything from // to the end of the line in which // appears.

A multiline comment occupies one or more lines of source code. This comment begins with the /* character sequence, continues with explanatory text, and ends with the */ character sequence. Everything from /* through */ is ignored by the compiler.

A Javadoc comment occupies one or more lines of source code. This comment begins with the /** character sequence, continues with explanatory text, and ends with the */ character sequence. Everything from /** through */ is ignored by the compiler.

Identifiers are used to name classes, methods, and other source code entities. An identifier consists of letters (A-Z, a-z, or equivalent uppercase/lowercase letters in other human alphabets), digits (0-9 or equivalent digits in other human alphabets), connecting punctuation characters (such as the underscore), and currency symbols (such as the dollar sign, $). This name must begin with a letter, a currency symbol, or a connecting punctuation character; and its length cannot exceed the line in which it appears. Some identifiers are reserved by Java. Examples include abstract and case.
Applications process different types of values such as integers, floating-point values, characters, and strings. A type identifies a set of values (and their representation in memory) and a set of operations that transform these values into other values of that set.

A primitive type is a type that’s defined by the language and whose values are not objects. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double precision floating-point primitive types.

A user-defined type is a type that’s defined by the developer using a class, an interface, an enum, or an annotation type and whose values are objects. User-defined types are also known as reference types.

An array type is a reference type that signifies an array, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as elements. This type consists of the element type and one or more pairs of square brackets that indicate the number of dimensions.

Applications manipulate values that are stored in memory, which is symbolically represented in source code through the use of the variables feature. A variable is a named memory location that stores some type of value.

Java provides the expressions feature for initializing variables and for other purposes. An expression combines some arrangement of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression’s type.

A simple expression is a literal, a variable name (containing a value), or a method call (returning a value). Java supports several kinds of literals: string, Boolean true and false, character, integer, floating-point, and null.

A compound expression is a sequence of simple expressions and operators, where an operator (a sequence of instructions symbolically represented in source code) transforms its operand expression value(s) into another value.

Java supplies many operators, which are classified by the number of operands that they take. A unary operator takes only one operand, a binary operator takes two operands, and Java’s single ternary operator takes three operands.

Operators are also classified as prefix, postfix, and infix. A prefix operator is a unary operator that precedes its operand, a postfix operator is a unary operator that trails its operand, and an infix operator is a binary or ternary operator that’s sandwiched between its operands.

Statements are the workhorses of a program. They assign values to variables, control a program’s flow by making decisions and/or repeatedly executing other statements, and perform other tasks. A statement can be expressed as a simple statement or as a compound statement.

In Chapter 3, I continue to explore the Java language by examining its support for classes and objects. You also learn more about arrays.
Discovering Classes and Objects

In Chapter 2, I introduced you to the fundamentals of the Java language. You now know how to write simple applications by inserting statements into a class's `main()` method. However, when you try to develop complex applications in this manner, you’re bound to find development tedious, slow, and prone to error. Classes and objects address these problems by simplifying application architecture.

In this chapter, I will introduce you to Java’s support for classes and objects. You will learn how to declare classes, construct objects from classes, encapsulate fields and methods in classes, restrict access to fields and methods, initialize classes and objects to appropriate startup values, and remove objects that are no longer needed.

In Chapter 2, I introduced you to arrays. You learned about array types and how to declare array variables, and you discovered a simple way to create an array. However, Java also provides a more powerful and more flexible way to create arrays, which is somewhat similar to how objects are created. This chapter also extends Chapter 2’s array coverage by introducing you to this capability.

**Declaring Classes**

A class is a container for housing an application (as demonstrated in Chapters 1 and 2), and it is also a template for manufacturing objects, which I discuss later in this chapter.

You declare a class by minimally specifying reserved word `class` followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code), followed by a body. The body starts with an open brace character (`{`) and ends with a close brace (`}`). Sandwiched between these delimiters are various kinds of member declarations. Consider Listing 3-1.

**Listing 3-1. Declaring a Skeletal Image Class**

```java
class Image
{
   // various member declarations
}
```

Chapter 3

www.it-ebooks.info
Listing 3-1 declares a class named `Image`, which presumably describes some kind of image for displaying on the screen. By convention, a class’s name, which must be a valid Java identifier, begins with an uppercase letter. Furthermore, the first letter of each subsequent word in a multiword class name is capitalized. This is known as camel casing.

You can choose any name for the file containing Listing 3-1, as long as the file extension is `.java`; a Java source file must have this extension. However, if you prefixed this declaration with the reserved word `public` (see Listing 3-2), the filename would have to match the class name or else the compiler would report an error. (Public classes are accessible from beyond their packages; see Chapter 5.)

**Listing 3-2. Declaring a Public Skeletal SavingsAccount Class**

```java
public class SavingsAccount {
    // various member declarations
}
```

Unlike Listing 3-1, which can be stored in `Image.java`, `x.java`, or some other named file with a `.java` file extension, Listing 3-2 must be stored in a file named `SavingsAccount.java`.

You can declare multiple classes in the same file, which Listing 3-3 demonstrates.

**Listing 3-3. Declaring Three Classes in the Same File**

```java
class A {
    // various member declarations
}
declass B {
    // various member declarations
}
class C {
    // various member declarations
}
```

Listing 3-3 declares classes `A`, `B`, and `C`. You can store this listing in `A.java`, `B.java`, or `C.java` (case doesn’t matter), `D.java`, or any other named file with a `.java` file extension.

If a source file contains multiple class declarations, you can declare at most one of these classes to be a public class. For example, Listing 3-4 is a variation of Listing 3-3 where `B` is declared `public`. 
Listing 3-4. Declaring Three Classes Where One Class is Public in the Same File

class A
{
    // various member declarations
}

public class B
{
    // various member declarations
}

class C
{
    // various member declarations
}

Listing 3-4 must be stored in a file named B.java.

Classes and Applications

In Chapter 2, I discussed application structure in terms of a public static void main(String[] args) method declared in a class. However, an application can consist of multiple classes and each class can declare its own main() method, which is demonstrated in Listing 3-5.

Listing 3-5. Declaring Three Classes with Their Own main() Methods

class A
{
    public static void main(String[] args)
    {
        // statements to execute
    }
}

class B
{
    public static void main(String[] args)
    {
        // statements to execute
    }
}

class C
{
    public static void main(String[] args)
    {
        // statements to execute
    }
}
Suppose Listing 3-5 was stored in App.java. You would compile this source file as follows:

javac App.java

You could then run the main() methods by executing the following commands:

java A
java B
java C

Does class A describe the application? How about class B, or even class C? This is certainly confusing. When you’re building a multiclass application and you’re new to Java, it’s best to declare a main() method in only one of its classes. That class would serve as the application’s entry point.

### Constructing Objects

Image, SavingsAccount, A, B, and C are examples of user-defined types from which objects (class instances) can be created. You create these objects by using the new operator with a constructor, which is a block of code that’s declared in a class for constructing an object from that class by initializing it in some manner.

Object creation has the following syntax:

```
new constructor
```

The new operator allocates memory to store the object whose type is specified by constructor and then invokes (calls) constructor to initialize the object, which is stored in the heap (a region of memory for storing objects). When constructor ends, new returns a reference (a memory address or other identifier) to the object so that it can be accessed elsewhere in the application.

The constructor has the following syntax:

```
class_name(parameter_list)
{
   // statements to execute
}
```

Unlike in a method declaration (discussed later in this chapter), a constructor doesn’t begin with a return type because it cannot return a value to new, which calls the constructor. If a constructor could return an arbitrary value, how would Java return that value? After all, the new operator returns a reference to an object, and how could new also return a constructor value?

A constructor doesn’t have a name. Instead, you must specify the name of the class that declares the constructor. This name is followed by a round bracket-delimited parameter list, which is a comma-separated list of zero or more parameter declarations. A parameter is a constructor or method variable that receives an expression value passed to the constructor or method when it’s called. This expression value is known as an argument.
CHAPTER 3: Discovering Classes and Objects

Note The number of arguments passed to a constructor or method, or the number of operator operands, is known as the constructor’s, method’s, or operator’s *arity*.

Consider the following example:

```java
Image image = new Image();
```

The `new` operator allocates memory to store an `Image` object in the heap. It then invokes a constructor with no parameters—a *noargument constructor*—to initialize this object. Following initialization, `new` returns a reference to the newly-initialized `Image` object. The reference is stored in a variable named `image` whose type is specified as `Image`. (It’s common to refer to the variable as an object, as in the `image` object, although it stores only an object’s reference and not the object itself.)

Note `new`’s returned reference is represented in source code by reserved word `this`. Wherever `this` appears, it represents the current object. Also, variables that store references are called *reference variables*.

**Default Constructor**

`Image` doesn’t explicitly declare a constructor. When a class doesn’t declare a constructor, Java implicitly creates a constructor for that class. The created constructor is known as the *default noargument constructor* because no arguments (demonstrated shortly) appear between its ( and ) characters when the constructor is invoked.

Note Java doesn’t create a default noargument constructor when at least one constructor is declared.

**Explicit Constructors**

You can explicitly declare a constructor within a class’s body. For example, Listing 3-6 enhances Listing 3-1’s `Image` class by declaring a single constructor with an empty parameter list.

*Listing 3-6. Declaring an Image Class with a Single Constructor*

```java
class Image
{
    Image()
    {
        System.out.println("Image() called");
    }
}
```
Listing 3-6's Image class declares a noargument constructor for initializing an Image object. The declaration consists of a class named Image followed by round brackets. This constructor simulates default initialization. It does so by invoking System.out.println() to output a message signifying that it's been called.

As previously shown with the default noargument constructor, you would create an object from this class by executing a statement such as the following:

```java
Image image = new Image();
```

The constructor would execute the System.out.println() method call, which results in the following output:

```java
Image() called
```

You will often declare constructors with nonempty parameter lists. Listing 3-7 demonstrates this scenario by declaring an Image class with a pair of constructors.

**Listing 3-7. Declaring an Image Class with Two Constructors**

```java
class Image {
    Image(String filename) {
        this(filename, null);
        System.out.println("Image(String filename) called");
    }

    Image(String filename, String imageType) {
        System.out.println("Image(String filename, String imageType) called");
        if (filename != null) {
            System.out.println("reading " + filename);
            if (imageType != null)
                System.out.println("interpreting " + filename + " as storing a " +
                    imageType + " image");
        }
        // Perform other initialization here.
    }
}
```

Listing 3-7's Image class first declares an Image(String filename) constructor whose parameter list consists of a single parameter declaration—a variable's type followed by the variable's name. The java.lang.String parameter is named filename, signifying that this constructor obtains image content from a file.
Note Throughout this and the remaining chapters, I typically prefix the first use of a predefined type (such as String) with the package hierarchy in which the type is stored. For example, String is stored in the lang subpackage of the java package. I do so to help you learn where types are stored in the standard class library. I will have more to say about packages in Chapter 5.

Image(String filename) demonstrates that some constructors rely on other constructors to help them initialize their objects. This is done to avoid redundant code, which increases the size of an object and unnecessarily takes memory away from the heap that could be used for other purposes. For example, Image(String filename) relies on Image(String filename, String imageType) to read the file's image content into memory.

Although it appears otherwise, constructors don't have names (however, it's common to refer to a constructor by specifying the class name and parameter list). A constructor calls another constructor by using keyword this and a round bracket-delimited and comma-separated list of arguments. For example, Image(String filename) executes this(filename, null); to execute Image(String filename, String imageType).

Caution You must use keyword this to call another constructor; you cannot use the class's name, as in Image(). The this() constructor call (when present) must be the first code that's executed within the constructor. This rule prevents you from specifying multiple this() constructor calls in the same constructor. Finally, you cannot specify this() in a method; constructors can be called only by other constructors and during object creation. (I discuss methods later in this chapter.)

When present, the constructor call must be the first code that's specified within a constructor; otherwise, the compiler reports an error. For this reason, a constructor that calls another constructor can perform additional work only after the other constructor has finished. For example, Image(String filename) executes System.out.println("Image(String filename) called"); after the invoked Image(String filename, String imageType) constructor finishes.

The Image(String filename, String imageType) constructor declares an imageType parameter that signifies the kind of image stored in the file—a Portable Network Graphics (PNG) image, for example. Presumably, the constructor uses imageType to speed up processing by not examining the file's contents to learn the image format. When null is passed to imageType, as happens with the Image(String filename) constructor, Image(String filename, String imageType) examines file content to learn the format. If null was also passed to filename, Image(String filename, String imageType) wouldn't read the file but would presumably notify the code attempting to create the Image object of an error condition.
The following example shows you how to create two Image objects, calling the first constructor with argument "image.png" and the second constructor with arguments "image.png" and "PNG". Each object's reference is assigned to a reference variable named image, replacing the previously stored reference for the second object assignment:

```java
Image image = new Image("image.png");
image = new Image("image.png", "PNG");
```

These constructor calls result in the following output:

```
Image(String filename, String imageType) called
reading image.png
Image(String filename) called
Image(String filename, String imageType) called
reading image.png
interpreting image.png as storing a PNG image
```

In addition to declaring parameters, a constructor can also declare variables within its body to help it perform various tasks. For example, the previously presented `Image(String filename, String imageType)` constructor might create an object from a (hypothetical) File class that provides the means to read a file's contents. At some point, the constructor instantiates this class and assigns the instance's reference to a variable, as demonstrated by the following example:

```java
Image(String filename, String imageType)
{
    System.out.println("Image(String filename, String imageType) called");
    if (filename != null)
    {
        System.out.println("reading " + filename);
        File file = new File(filename);
        // Read file contents into object.
        if (imageType != null)
            System.out.println("interpreting " + filename + " as storing a " +
            imageType + " image");
        else
            // Inspect image contents to learn image type.
            ; // Empty statement is used to make if-else syntactically valid.
    }
    // Perform other initialization here.
}
```
As with the filename and imageType parameters, file is a variable that's local to the constructor, and it is known as a local variable to distinguish it from a parameter. Although all three variables are local to the constructor, there are two key differences between parameters and local variables:

- The filename and imageType parameters come into existence at the point where the constructor begins to execute and exist until execution leaves the constructor. In contrast, file comes into existence at its point of declaration and continues to exist until the block in which it's declared is terminated (via a closing brace character). This property of a parameter or a local variable is known as lifetime.

- The filename and imageType parameters can be accessed from anywhere in the constructor. In contrast, file can be accessed only from its point of declaration to the end of the block in which it's declared. It cannot be accessed before its declaration or after its declaring block, but nested sub-blocks can access the local variable. This property of a parameter or a local variable is known as scope.

Objects and Applications

You previously learned that you can declare a public static void main(String[] args) entry-point method in any class. Also, it can be confusing to declare this method in every class of a multiclass application because your application's users won’t know which class is the application entry point.

Sometimes, you'll want to declare main() in multiple classes for testing purposes. You can then test that class independently of the other classes that make up the application. Alternatively, you might decide to perform unit testing (http://en.wikipedia.org/wiki/Unit_testing) on your classes.

Listing 3-8 presents a three-class application that demonstrates multiclass testing.

Listing 3-8. Testing an Application's Component Classes

class Circle
{
    Circle()
    {
        System.out.println("Circle() called");
    }

    public static void main(String[] args)
    {
        new Circle();
    }
}

class Rectangle
{
    Rectangle()
    {
        System.out.println("Rectangle() called");
    }
}
Listing 3-8 declares three classes: Circle, Rectangle, and Shapes. Unlike Circle and Rectangle, Shapes is declared public. Because Shapes is public and all three classes are declared in the same source file, the name of this source file is Shapes.java.

Each class declares a main() method. However, the main application class is Shapes because it demonstrates Circle and Rectangle, which are components of the application. To identify an application’s main class in a multiclass application, I declare the main class public.

You would execute the following command to compile Shapes.java:

```
javac Shapes.java
```

To run the Shapes application, you would execute the following command:

```
java Shapes
```

You can also run the Circle and Rectangle component classes to test that these components work properly:

```
java Circle
java Rectangle
```

When you’re finished testing the component classes, you should probably remove the main() methods from them to avoid confusion on the part of anyone studying your code. Alternatively, you could document these methods via comments and clearly document the entry-point class.

## Encapsulating State and Behaviors

Classes typically combine state with behaviors. State refers to attributes (such as a counter set to 1 or an account balance storing $20,000) that are read and/or written when an application runs, and behaviors refer to sequences of code (such as calculate a specific factorial or make a deposit into a savings account) that read/write attributes and perform other tasks. Combining state with behaviors is known as encapsulation.

```java
public static void main(String[] args) {
    new Rectangle();
}

public class Shapes {
    public static void main(String[] args) {
        Circle c = new Circle();
        Rectangle r = new Rectangle();
    }
}
```
Representing State via Fields

Java represents state via *fields*, which are variables declared within a class’s body. State associated with a class is described by class fields, whereas state associated with objects is described by *object fields* (also known as *instance fields*).

**Note** By convention, a field’s name begins with a lowercase letter, and the first letter of each subsequent word in a multiword field name is capitalized.

Declaring and Accessing Class Fields

A class field stores an attribute that’s associated with a class. All objects created from that class share this class field. When one object modifies the field’s value, the new value is visible to all current and future objects created from that class.

You declare a class field by specifying the following syntax:

```java
static type_name variable_name [ = expression ] ;
```

A class field declaration begins with the `static` reserved word. This reserved word is followed by a type name and a variable name. You can optionally end the declaration by assigning a type-compatible expression, which is known as a *class field initializer*, to the variable name. Don’t forget to specify the trailing semicolon character.

For example, suppose you declare a `Car` class and want to keep track of the number of objects that are created from this class. To accomplish this task, you introduce a `counter` class field (initialized to 0) into this class. Check out Listing 3-9.

**Listing 3-9. Adding a counter Class Field to a Car Class**

```java
class Car
{
    static int counter = 0;

    Car()
    {
        counter++;
    }
}
```

Listing 3-9 declares an `int` class field named `counter`. The `static` prefix implies that there is only one copy of this field and not one copy per object. This `counter` field is explicitly initialized to 0. Each time an object is created, the `counter++` expression in the `Car()` constructor increases `counter` by 1.

Listing 3-10 presents a `Cars` application class that demonstrates `Car` and `counter`.
Listing 3-10. Demonstrating the Car Class and Its counter Field

```java
public class Cars {
    public static void main(String[] args) {
        System.out.println(Car.counter);
        Car myCar = new Car();
        System.out.println(Car.counter);
        Car yourCar = new Car();
        System.out.println(Car.counter);
    }
}
```

Compile Listing 3-10 as follows. Listing 3-9 is also compiled because Cars references Car.

```bash
javac Cars.java
```

Run the Cars application as follows:

```bash
java Cars
```

You should observe the following output, which reveals counter's initial value and that this variable is incremented for each created Car object:

```
0
1
2
```

It isn't necessary to explicitly initialize counter to 0. When a class is loaded into memory, class fields are initialized to default zero/false/null values. For example, counter is implicitly initialized to 0.

**Note**  Class fields are initialized by zeroing their bits. You interpret this value as literal false, '\u0000', 0.0, 0.0f, 0, 0L, or null depending on the field's type.

Within a class declaration, a class field is accessed directly, as in counter++. When accessing a class field from outside of the class, you must prepend the class name followed by the member access operator to the class field name. For example, to access counter from Cars, you must specify Car.counter.

**Note**  I could have accessed counter via the myCar and yourCar reference variables, for example, myCar.counter and yourCar.counter. However, the preferred approach is to use the class name, as in Car.counter. This approach is preferred because it's easier to tell that a class field is being accessed.
Class fields can be modified. If you want a class field to be constant (an unchangeable variable), you must declare the class field to be final by prefixing its declaration with the final reserved word. Listing 3-11 presents an example.

Listing 3-11. Declaring a Constant in the Employee class

class Employee
{
    final static int RETIREMENT_AGE = 65;
}

Listing 3-11 declares an integer constant named RETIREMENT_AGE. If you attempt to modify this variable subsequently, as in RETIREMENT_AGE = 32;, the compiler reports an error. Although reserved word final preceeds reserved word static in this example, you can switch this order, which results in static final int RETIREMENT_AGE = 65;.

The RETIREMENT_AGE declaration is an example of a compile-time constant. Because there is only one copy of its value (because of static), and because this value will never change (thanks to final), the compiler can optimize the bytecode by inserting the constant value into all calculations where it appears. Code runs faster because it doesn’t have to access a read-only class field.

A class field is created when the class that declares this field is loaded into the heap. The class field is destroyed when the class is unloaded, typically when the virtual machine ends. This property of a class field is known as lifetime.

A class field is visible to the entire class in which it’s declared. In other words, the class field can be accessed from anywhere in its class. Unless explicitly hidden (discussed later in this chapter), the class field is visible to code outside of the class. This property of a class field is known as scope.

Declaring and Accessing Instance Fields

An instance field stores an attribute that’s associated with an object. Each object maintains a separate copy of the attribute. For example, one object might have red as its color attribute, whereas a second object has green as its color attribute.

You declare an instance field by specifying the following syntax:

type_name variable_name [ = expression ];

An instance field declaration begins with a type name followed by a variable name. You can optionally end the declaration by assigning a type-compatible expression, which is known as an instance field initializer, to the variable name. Don’t forget to specify the trailing semicolon character.

For example, you want to model a car in terms of its make, model, and number of doors. You don’t use class fields to store these attributes because cars have different makes, models, and door counts. Listing 3-12 presents a Car class with three instance field declarations for these attributes.
Listing 3-12. Declaring a Car Class with make, model, and numDoors Instance Fields

class Car
{
    String make;
    String model;
    int numDoors;
}

Listing 3-12 declares two String instance fields named make and model. It also declares an int instance field named numDoors.

When an object is created, instance fields are initialized to default zero values, which you interpret at the source code level as literal value false, '\u0000', 0, 0L, 0.0, 0.0F, or null (depending on field type). For example, if you executed Car car = new Car();, make and model would be initialized to null and numDoors would be initialized to 0.

Listing 3-13 presents the source code to a Cars application class that directly accesses its instance fields.

Listing 3-13. Directly Accessing Instance Fields

public class Cars
{
    public static void main(String[] args)
    {
        Car myCar = new Car();
        myCar.make = "Toyota";
        myCar.model = "Camry";
        myCar.numDoors = 4;
        System.out.println("Make = " + myCar.make);
        System.out.println("Model = " + myCar.model);
        System.out.println("Number of doors = " + myCar.numDoors);
    }
}

Compile Listing 3-13 as follows:

javac Cars.java

Run the Cars application as follows:

java Cars

You should observe the following output, which reveals the values assigned to the Car object's make, model, and numDoors instance fields:

Make = Toyota
Model = Camry
Number of doors = 4
In a class declaration, an instance field is accessed directly, as in System.out.println(numDoors);
When accessing an instance field from outside of an object, you must prepend the desired object
reference variable followed by the member access operator to the instance field name. For example,
I specified myCar.make to access the make field of the myCar object in Listing 3-13.

You can explicitly initialize an instance field when declaring that field to provide a nonzero default
value, which overrides the default zero value. Listing 3-14 demonstrates this point.

Listing 3-14. Initializing Car’s numDoors Instance Field to a Default Nonzero Value

class Car
{
    String make;
    String model;
    int numDoors = 4;
}

Listing 3-14 explicitly initializes numDoors to 4 because the developer has assumed that most cars
being modeled by this class have four doors. When Car is initialized via the default noargument
Car() constructor, which is responsible for assigning 4 to numDoors, the developer only needs to
initialize the make and model instance fields for those cars that have four doors.

You could remove the myCar.numDoors = 4; assignment from Listing 3-13, and you would still
observe the same output.

It’s usually not a good idea to directly initialize an object’s instance fields, and you will learn why
when I discuss information hiding later in this chapter. Instead, you should perform this initialization
in the class’s constructor(s); see Listing 3-15.

Listing 3-15. Initializing Car’s Instance Fields via Constructors

class Car
{
    String make;
    String model;
    int numDoors;

    Car(String make, String model)
    {
        this(make, model, 4);
    }

    Car(String make, String model, int nDoors)
    {
        this.make = make;
        this.model = model;
        numDoors = nDoors;
    }
}
Listing 3-15’s Car class declares Car(String make, String model) and Car(String make, String model, int nDoors) constructors. The first constructor lets you specify make and model, whereas the second constructor lets you specify values for the three instance fields.

The first constructor executes this(make, model, 4); to pass the values of its make and model parameters, along with a default value of 4 to the second constructor. Doing so demonstrates an alternative to initializing an instance field explicitly, and it is preferable from a code maintenance perspective.

The Car(String make, String model, int numDoors) constructor demonstrates another use for keyword this. Specifically, it demonstrates a scenario where constructor parameters have the same names as the class’s instance fields. Prefixing a variable name with this. causes the Java compiler to create bytecode that accesses the instance field. For example, this.make = make; assigns the make parameter’s String object reference to this (the current) Car object’s make instance field. If make = make; was specified instead, it would accomplish nothing by assigning make’s value to itself; a Java compiler might not generate code to perform the unnecessary assignment. In contrast, this. isn’t necessary for the numDoors = nDoors; assignment, which initializes the numDoors field from the nDoors parameter value.

Note To minimize error (by forgetting to prefix a field name with this.), it’s preferable to keep field names and parameter names distinct (such as numDoors and nDoors). Alternatively, you might prefix a field name with an underscore (such as _nDoors). Either way, you wouldn’t have to worry about the this. prefix (and forgetting to specify it).

Listing 3-16 demonstrates instance field initialization via these constructors.

Listing 3-16. Demonstrating Instance Field Initialization via Constructors

```java
public class Cars {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry");
        System.out.println("Make = " + myCar.make);
        System.out.println("Model = " + myCar.model);
        System.out.println("Number of doors = " + myCar.numDoors);
        System.out.println();
        Car yourCar = new Car("Mazda", "RX-8", 2);
        System.out.println("Make = " + yourCar.make);
        System.out.println("Model = " + yourCar.model);
        System.out.println("Number of doors = " + yourCar.numDoors);
    }
}
```

www.it-ebooks.info
CHAPTER 3: Discovering Classes and Objects

Compile Listing 3-16 as follows:

```java
javac Cars.java
```

Run the `Cars` application as follows:

```java
java Cars
```

You should observe the following output, which reveals that each of the `myCar` and `yourCar` objects has different instance field values:

```plaintext
Make = Toyota
Model = Camry
Number of doors = 4

Make = Mazda
Model = RX-8
Number of doors = 2
```

Instance fields can be modified. If you want an instance field to be constant, you must declare the instance field to be final by prefixing its declaration with the `final` reserved word. Listing 3-17 presents an example.

**Listing 3-17. Declaring a Constant in the `Employee` class**

```java
class Employee {
    final int RETIREMENT_AGE = 65;
}
```

Listing 3-17 declares an integer constant named `RETIREMENT_AGE`. If you attempt to modify this variable subsequently, which you can only do in an object context (such as `emp.RETIREMENT_AGE = 32;`), the compiler reports an error.

Each object receives a copy of a read-only instance field. Regarding Listing 3-17, each `Employee` object would receive a copy of `RETIREMENT_AGE`. Because this is wasteful of memory, you would be better off also declaring `RETIREMENT_AGE` to be static so that there is only one copy.

A final instance field must be initialized, as part of the field’s declaration or in the class’s constructor. When initialized in the constructor, the read-only instance field is known as a `blank final` because it doesn’t have a value (the field is blank) until one is assigned to it in the constructor. Because a constructor can potentially assign a different value to each object’s blank final, these read-only variables are only true constants in the contexts of their objects. Check out Listing 3-18.

**Listing 3-18. Declaring a Different Constant for Each `Employee` class**

```java
class Employee {
    final int ID;
    static int counter;
}
```
Employee()
{
    ID = counter++;
}

Each Employee object receives a different read-only ID value.

An instance field is created when an object is created from the class that declares this field. The instance field is destroyed when the object is garbage collected. (I discuss garbage collection later in this chapter.) This property of an instance field is known as lifetime.

An instance field is visible to the entire class in which it’s declared. In other words, the instance field can be accessed from anywhere in its class. Unless explicitly hidden (discussed later in this chapter), the instance field is visible to code outside of the class, but only in the context of an object reference variable. This property of an instance field is known as scope.

**Reviewing Field-Access Rules**

The previous examples of field access may seem confusing because you can sometimes specify the field’s name directly, whereas you need to prefix a field name with an object reference or a class name and the member access operator at other times. The following rules dispel this confusion by giving you guidance on how to access fields from the various contexts:

- Specify the name of a class field as is from anywhere within the same class as the class field declaration. Example: `counter`
- Specify the name of a class field’s class, followed by the member access operator, followed by the name of the class field from outside the class. Example: `Car.counter`
- Specify the name of an instance field as is from any instance method, constructor, or instance initializer (discussed later) in the same class as the instance field declaration. Example: `numDoors`
- Specify an object reference, followed by the member access operator, followed by the name of the instance field from any class method or class initializer (discussed later) within the same class as the instance field declaration or from outside the class. Example: `Car car = new Car(); car.numDoors = 2;`

Although the final rule might seem to imply that you can access an instance field from a class context, this isn’t the case. Instead, you’re accessing the field from an object context.

The previous access rules aren’t exhaustive because there are two more field-access scenarios to consider: declaring a parameter or local variable with the same name as an instance field or as a class field. In either scenario, the local variable/parameter is said to shadow (hide or mask) the field.

If you’ve declared a parameter/local variable that shadows a field, you can rename it, or you can use the member access operator with a class name (class field) or reserved word `this` (instance field) to identify the field explicitly. For example, Listing 3-15’s `Car(String make, String model, int nDoors)` constructor demonstrated this latter solution by specifying statements such as `this.make = make;` to distinguish an instance field from a same-named parameter/local variable.
Representing Behaviors via Methods

Java represents behaviors via methods, which are named blocks of code declared within a class’s body. Behaviors associated with a class are described by class methods, whereas behaviors associated with objects are described by object methods (also known as instance methods).

Note By convention, a method’s name begins with a lowercase letter, and the first letter of each subsequent word in a multiword method name is capitalized.

Declaring and Invoking Class Methods

A class method stores a behavior that’s associated with a class. All objects created from that class share this class method. The class method has no direct access to instance fields. The only way to access these fields is to access them in the context of a specific object, via an object reference variable and the member access operator.

A class method has the following syntax:

```
static return_type name(parameter_list)
{
    // statements to execute
}
```

A class method begins with a header that starts with static. This reserved word is followed by a return type that specifies the type of value that the method returns. The return type is either a primitive type name (such as int or double), a reference type name (such as String), or reserved word void when the class method doesn’t return any kind of value.

The class method’s name follows its return type. This name must be a legal Java identifier that isn’t a reserved word.

Note A method’s name and the number, types, and order of its parameters are known as its signature.

As with a constructor, the class method header concludes with a parameter list that lets you specify the kinds of data items that are passed to the method for processing.

The header is followed by a brace-delimited body of statements that execute when the class method is invoked.

You’ve already encountered the public static void main(String[] args) class method that serves as a class’s entry point. The java tool creates an array of String objects, one object per command-line argument, and passes this array to the args parameter so that code within main() can process these arguments when it invokes main().

For a second class method example, consider Listing 3-19’s Utilities class and its dumpMatrix() class method.
Listing 3-19. A Utilities Class with a Single Class Method for Dumping a Matrix in Tabular Format

```java
public class Utilities {
    static void dumpMatrix(float[][] matrix) {
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++) {
                System.out.print(matrix[row][col] + " ");
                System.out.print("\n");
            }
        }
    }

    public static void main(String[] args) {
        float[][] temperatures = {
            { 37.0f, 14.0f, -22.0f },
            { 0.0f, 29.0f, -5.0f }
        };
        dumpMatrix(temperatures);
        System.out.println();
        Utilities.dumpMatrix(temperatures);
    }
}
```

The `dumpMatrix()` class method dumps the contents of a two-dimensional array in tabular format to the standard output stream. The method's header tells us that the method doesn't return anything (its return type is `void`), the method is named `dumpMatrix`, and the method contains a parameter list consisting of a single parameter named `matrix`, which is of type `float[][]`.

Within the method, `row` and `col` are declared as local variables. The `row` variable is declared in the outer `for` loop. Its scope ranges from its `for` loop header through the end of the brace-delimited block that follows this header. The `col` variable is declared in the inner `for` loop. Its scope ranges from its `for` loop header through the end of the single-statement block (`System.out.print(matrix[row][col] + " ");`) that follows this header.

A third class method, `public static void main(String[] args)`, is present for testing purposes. After declaring a `temperatures` matrix, it shows you two ways to invoke `dumpMatrix()`. The invocation without a prefix is used when the class method being invoked is declared in the same class. A prefix is typically specified only when the class method is being called from a different class.

Compile Listing 3-19 as follows:

```
javac Utilities.java
```

Run the `Utilities` application as follows:

```
java Utilities
```
You should observe the following output:

```
37.0 14.0 -22.0
0.0 29.0 -5.0
37.0 14.0 -22.0
0.0 29.0 -5.0
```

**METHOD-CALL STACK**

Method invocations require a method-call stack (also known as a method-invocation stack) to keep track of the statements to which execution must return. Think of the method-call stack as a simulation of a pile of clean trays in a cafeteria. You pop (remove) the clean tray from the top of the pile and the dishwasher will push (insert) the next clean tray onto the top of the pile.

When a method is invoked, the virtual machine pushes its arguments and the address of the first statement to execute following the invoked method onto the method-call stack. The virtual machine also allocates stack space for the method’s local variables. When the method returns, the virtual machine removes local variable space, pops the address and arguments off of the stack, and transfers execution to the statement at this address.

### Declaring and Invoking Instance Methods

An instance method stores a behavior that’s associated with an object. Unlike a class method, an instance method can directly access an object’s instance fields.

An instance method has the following syntax:

```
return_type name(parameter_list)
{
   // statements to execute
}
```

Apart from the absence of the static reserved word, this syntax is the same as the syntax for a class method.

Listing 3-20 refactors Listing 3-15’s Car class also to include a printDetails() instance method.

*Listing 3-20. Extending the Car class with an Instance Method that Prints Details*

```java
class Car
{
   String make;
   String model;
   int numDoors;
```
Car(String make, String model)
{
    this(make, model, 4);
}

Car(String make, String model, int nDoors)
{
    this.make = make;
    this.model = model;
    numDoors = nDoors;
}

void printDetails()
{
    System.out.println("Make = " + make);
    System.out.println("Model = " + model);
    System.out.println("Number of doors = " + numDoors);
    System.out.println();
}

The printDetails() method has its return type set to void because it doesn’t return a value. It prints out the car’s make, model, and number of doors; and then outputs a blank line separator.

Listing 3-21 presents a Cars application that creates Car objects and invokes each object’s printDetails() instance method.

Listing 3-21. Demonstrating Instance Method Calls

public class Cars
{
    public static void main(String[] args)
    {
        Car myCar = new Car("Toyota", "Camry");
        myCar.printDetails();
        Car yourCar = new Car("Mazda", "RX-8", 2);
        yourCar.printDetails();
    }
}

The main() method instantiates Car, passing appropriate make and model strings to its two-parameter constructor; the number of doors defaults to 4. It then invokes printDetails() on the returned object reference to print these values. Next, main() creates a second Car object via the three-parameter constructor and prints out this object’s make, model, and number of doors.

main()’s invocation of printDetails() demonstrates that an instance method is always invoked in an object reference context.

Compile Listing 3-21 as follows:

javac Cars.java
Run the Cars application as follows:

```
java Cars
```

You should observe the following output:

```
Make = Toyota  
Model = Camry 
Number of doors = 4

Make = Mazda  
Model = RX-8 
Number of doors = 2
```

**Note**  When an instance method is invoked, Java passes a hidden argument to the method (as the leftmost argument in a list of arguments). This argument is the reference to the object on which the method is invoked. It's represented at the source code level via reserved word `this`. You don’t need to prefix an instance field name with “this.” from within the method whenever you attempt to access an instance field name that isn’t also the name of a parameter because the Java compiler ensures that the hidden argument is used to access the instance field.

**Returning from a Method via the Return Statement**

Java provides the return statement to terminate method execution and return control to the method’s caller (the code sequence that called the method). This statement has the following syntax:

```
return [ expression ] ;
```

You can specify `return` without an expression and will typically use this form of the return statement to return prematurely from a method—or from a constructor. In other words, you don’t want to execute all of the statements in the method/constructor. Check out Listing 3-22.

**Listing 3-22. Returning Prematurely from an Instance Method**

```
public class Employee
{
  String name;

  Employee(String name)
  {
    setName(name);
  }
```
void setName(String name)
{
    if (name == null)
    {
        System.out.println("name cannot be null");
        return;
    }
    else
        this.name = name;
}

public static void main(String[] args)
{
    Employee john = new Employee(null);
}

Listing 3-22’s Employee(String name) constructor invokes the setName() instance method to initialize the name instance field. Providing a separate method for this purpose is a good idea because it lets you initialize the instance field at construction time and also at a later time. (Perhaps the employee changes his or her name.)

setName() uses an if statement to detect an attempt to assign the null reference to the name field. When such an attempt is detected, it outputs the "name cannot be null" error message and returns prematurely from the method so that the null value cannot be assigned (and replace a previously assigned name).

Compile Listing 3-22 as follows:

javalv Employee.java

Run the Employee application as follows:

java Employee

You should observe the following output:

name cannot be null

Caution When using the return statement, you might run into a situation where the compiler outputs an "unreachable code" error message. It does so when it detects code that will never be executed and occupies memory unnecessarily. One area where you might encounter this problem is the switch statement. For example, suppose you specify case 2: printUsageInstructions(); return; break; as part of this statement. The compiler reports an error when it detects the break statement following the return statement because the break statement is unreachable; it never can be executed.
The previous form of the return statement isn't legal in a method that returns a value. For such methods, Java lets the method return a value (whose type must be compatible with the method's return type). The following example demonstrates this version:

```java
static double divide(double dividend, double divisor)
{
    if (divisor == 0.0)
    {
        System.out.println("cannot divide by zero");
        return 0.0;
    }
    return dividend / divisor;
}
```

divide() uses an if statement to detect an attempt to divide its first argument by 0.0 and outputs an error message when this attempt is detected. Furthermore, it returns 0.0 to signify this attempt. If there is no problem, the division is performed and the result is returned.

**Caution** You cannot use this form of the return statement in a constructor because constructors don’t have return types.

### Chaining Together Instance Method Calls

Two or more instance method calls can be chained together via the member access operator, which results in more compact code. To accomplish instance method call chaining, you need to rearchitect your instance methods somewhat differently, which Listing 3-23 reveals.

**Listing 3-23. Implementing Instance Methods So That Calls to These Methods Can be Chained Together**

```java
public class SavingsAccount
{
    int balance;

    SavingsAccount deposit(int amount)
    {
        balance += amount;
        return this;
    }

    SavingsAccount printBalance()
    {
        System.out.println(balance);
        return this;
    }
}```
public static void main(String[] args) {
    new SavingsAccount().deposit(1000).printBalance();
}

To achieve instance method call chaining, Listing 3-23 declares SavingsAccount as the return type for the deposit() and printBalance() methods. Also, each method specifies return this; (return current object’s reference) as its last statement.

In the main() method, new SavingsAccount().deposit(1000).printBalance(); performs the following tasks.

1. It creates a SavingsAccount object.
2. It uses the returned SavingsAccount reference to invoke SavingsAccount’s deposit() instance method, to add one thousand dollars to the savings account (I’m ignoring cents for convenience).
3. It uses deposit()’s returned SavingsAccount reference to invoke SavingsAccount’s printBalance() instance method to output the account balance.

Compile Listing 3-23 as follows:

cjavac SavingsAccount.java

Run the SavingsAccount application as follows:

cjava SavingsAccount

You should observe the following output:

1000

**Note** I’ll have more to say about instance method call chaining when I discuss fluent interfaces in Chapter 16.

**Passing Arguments to Methods**

A method call includes a list of (zero or more) arguments being passed to the method. Java passes arguments to methods via a style of argument passing called *pass-by-value*, which the following example demonstrates:

```java
Employee emp = new Employee("John ");
int recommendedAnnualSalaryIncrease = 1000;
printReport(emp, recommendAnnualSalaryIncrease);
printReport(new Employee("Cuifen"), 1500);
```
Pass-by-value passes the value of a variable (the reference value stored in emp or the 1000 value stored in recommendedAnnualSalaryIncrease, for example) or the value of some other expression (such as new Employee("Cuifen") or 1500) to the method.

Because of pass-by-value, you cannot assign a different Employee object’s reference to emp from inside printReport() via the printReport() parameter for this argument. After all, you have only passed a copy of emp’s value to the method.

Many methods and constructors require you to pass a fixed number of arguments when they are called. However, Java also provides the ability to pass a variable number of arguments; such methods/constructors are often referred to as varargs methods/constructors. To declare a method or constructor that takes a variable number of arguments, specify three consecutive periods after the type name of the method’s/constructor’s rightmost parameter.

The following example presents a sum() method that accepts a variable number of arguments:

```java
static double sum(double... values)
{
    int total = 0;
    for (int i = 0; i < values.length; i++)
        total += values[i];
    return total;
}
```

sum()'s implementation totals the number of arguments passed to this method, for example, sum(10.0, 20.0) or sum(30.0, 40.0, 50.0). (Behind the scenes, these arguments are stored in a one-dimensional array, as evidenced by values.length and values[i].) After these values have been totaled, this total is returned via the return statement.

**Invoking Methods Recursively**

A method normally executes statements that may include calls to other methods, such as printDetails() invoking System.out.println(). However, it’s occasionally convenient to have a method call itself. This scenario is known as recursion.

For example, suppose you need to write a method that returns a factorial (the product of all the positive integers up to and including a specific integer). For example, 3! (the ! is the mathematical symbol for factorial) equals 3×2×1 or 6.

Your first approach to writing this method might consist of the code presented in the following example:

```java
static int factorial(int n)
{
    int product = 1;
    for (int i = 2; i <= n; i++)
        product *= i;
    return product;
}
```
Although this code accomplishes its task (via iteration), `factorial()` could also be written according to the following example’s recursive style:

```java
static int factorial(int n)
{
    if (n == 1)
        return 1; // base problem
    else
        return n * factorial(n - 1);
}
```

The recursive approach takes advantage of being able to express a problem in simpler terms of itself. According to this example, the simplest problem, which is also known as the base problem, is $1! (1)$.

When an argument greater than 1 is passed to `factorial()`, this method breaks the problem into a simpler problem by calling itself with the next smaller argument value. Eventually, the base problem will be reached.

For example, calling `factorial(4)` results in the following stack of expressions:

- $4 \times \text{factorial}(3)$
- $3 \times \text{factorial}(2)$
- $2 \times \text{factorial}(1)$

This last expression is at the top of the stack. When `factorial(1)` returns 1, these expressions are evaluated as the stack begins to unwind:

- $2 \times \text{factorial}(1)$ now becomes $2 \times 1 = 2 (2)$
- $3 \times \text{factorial}(2)$ now becomes $3 \times 2 = 6 (6)$
- $4 \times \text{factorial}(3)$ now becomes $4 \times 6 = 24 (24)$

Recursion provides an elegant way to express many problems. Additional examples include searching tree-based data structures for specific values and, in a hierarchical file system, finding and outputting the names of all files that contain specific text.

**Caution** Recursion consumes stack space, so make sure that your recursion eventually ends in a base problem; otherwise, you will run out of stack space and your application will be forced to terminate.

**Overloading Methods**

Java lets you introduce methods with the same name but different parameter lists into the same class. This feature is known as method overloading. When the compiler encounters a method invocation expression, it compares the called method’s arguments list with each overloaded method’s parameter list as it looks for the correct method to invoke.
Two same-named methods are overloaded when their parameter lists differ in number or order of parameters. Alternatively, two same-named methods are overloaded when at least one parameter differs in type. Listing 3-24 presents an application that demonstrates these scenarios in an instance method context.

Listing 3-24. Demonstrating Method Overloading

```java
public class MO {
    int add(int a, int b) {
        System.out.println("add(int, int) called");
        return a + b;
    }
    int add(int a, int b, int c) {
        System.out.println("add(int, int, int) called");
        return a + b + c;
    }
    double add(double a, double b) {
        System.out.println("add(double, double) called");
        return a + b;
    }
    public static void main(String[] args) {
        MO mo = new MO();
        int result = mo.add(10, 20);
        System.out.println("Result = " + result);
        result = mo.add(10, 20, 30);
        System.out.println("Result = " + result);
        double result2 = mo.add(5.0, 8.0);
        System.out.println("Result2 = " + result2);
    }
}
```

After creating an MO object, main() invokes this object's int add(int a, int b) instance method to add the two integer arguments together and save the result, which is subsequently output. Next, main() invokes int add(int a, int b, int c) to add the three integer arguments together and outputs the result. Finally, main() invokes double add(double a, double b) to add the two double precision floating-point arguments together and outputs the result.

Compile Listing 3-24 as follows:

```
javac MO.java
```
Run the M0 application as follows:

java M0

You should observe the following output:

<table>
<thead>
<tr>
<th>Method Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(int, int) called</td>
<td>Result = 30</td>
</tr>
<tr>
<td>add(int, int, int) called</td>
<td>Result = 60</td>
</tr>
<tr>
<td>add(double, double) called</td>
<td>Result2 = 13.0</td>
</tr>
</tbody>
</table>

You cannot overload a method by changing only the return type. For example, `double sum(double... values)` and `int sum(double... values)` are not overloaded. These methods are not overloaded because the compiler doesn’t have enough information to choose which method to call when it encounters `sum(1.0, 2.0)` in source code.

### Reviewing Method-Invocation Rules

The previous examples of method invocation may seem confusing because you can sometimes specify the method’s name directly, whereas you need to prefix a method name with an object reference or a class name and the member access operator at other times. The following rules dispel this confusion by giving you guidance on how to invoke methods from the various contexts:

- **Specify the name of a class method as is from anywhere within the same class as the class method.** Example: `dumpMatrix(temperatures);`

- **Specify the name of the class method’s class, followed by the member access operator, followed by the name of the class method from outside the class.** Example: `Utilities.dumpMatrix(temperatures);` (You can also invoke a class method via an object instance, but that is considered bad form because it hides from casual observation the fact that a class method is being invoked.)

- **Specify the name of an instance method as is from any instance method, constructor, or instance initializer in the same class as the instance method.** Example: `setName(name);`

- **Specify an object reference, followed by the member access operator, followed by the name of the instance method from any class method or class initializer within the same class as the instance method or from outside the class.** Example: `Car car = new Car("Toyota", "Camry"); car.printDetails();`

Although the latter rule might seem to imply that you can call an instance method from a class context, this isn’t the case. Instead, you call the method from an object context.

Also, don’t forget to make sure that the number of arguments passed to a method, along with the order in which these arguments are passed, and the types of these arguments agree with their parameter counterparts in the method being invoked.
CHAPTER 3: Discovering Classes and Objects

Note  Field access and method call rules are combined in expression System.out.println();, where the leftmost member access operator accesses the out class field (of type java.io.PrintStream) in the java.lang.System class, and where the rightmost member access operator calls this field's println() method. You’ll learn about PrintStream in Chapter 11 and System in Chapter 7.

Hiding Information

Every class X exposes an interface, which are the constructors, methods, and (possibly) fields that can be accessed from outside of X. For example, in Listing 3-9, class field counter can be accessed from outside of its containing Car class so counter is part of that class's interface. Also, in Listing 3-20, instance method printDetails() can be accessed from outside of its containing Car class so printDetails() is part of that class's interface.

An interface serves as a contract between a class and its clients, which are external classes that communicate with the class and/or its instances by accessing fields (typically public static final fields, or constants) and calling constructors and methods. The contract is such that the class promises to not change its interface, which would break dependent clients.

X also provides an implementation (the code within exposed methods along with optional helper methods and optional supporting fields that shouldn’t be exposed) that codifies the interface. Helper methods are methods that assist exposed methods and shouldn’t be exposed themselves.

When designing a class, your goal is to expose a useful interface while hiding details of that interface’s implementation. For example, consider Listing 3-15’s Car class. This class exposes make, model, and numDoors instance fields along with a pair of constructors. Many developers would regard these instance fields to belong to Car’s implementation and should be hidden. After all, they could be renamed, their types could be changed, or they could be removed in a future version of the class, which would break dependent client code.

Note  In contrast to nonconstant fields, you often expose constant fields. For example, Listing 3-11’s Employee class exposes a RETIREMENT_AGE constant class field. However, you would probably hide constant fields that are only relevant within the context of the class in which they are declared. You would do so to prevent them from cluttering up the class’s interface and exposing clients to unnecessary details.

You hide the implementation to prevent developers from accidentally accessing parts of your class that don’t belong to the class’s interface so that you’re free to change the implementation without breaking client code. Hiding the implementation is often referred to as information hiding. Furthermore, many developers consider implementation hiding to be part of encapsulation.
Java supports implementation hiding by providing four levels of access control, where three of these levels are indicated via a reserved word. You can use the following access control levels to control access to fields, methods, and constructors and two of these levels to control access to classes:

- **Public**: A field, method, or constructor that is declared `public` is accessible from anywhere. Classes can be declared `public` as well. I typically declare a class that contains the `public static void main(String[] args)` entry-point method `public`. Classes that are to be visible outside their packages (see Chapter 5) are also declared `public`. Public classes must be declared in files whose names match the class names.

- **Protected**: A field, method, or constructor that is declared `protected` is accessible from all classes in the same package as the member's class as well as subclasses of that class regardless of package.

- **Private**: A field, method, or constructor that is declared `private` cannot be accessed from outside the class in which it's declared.

- **Package-private**: In the absence of an access-control reserved word, a field, method, or constructor is only accessible to classes within the same package as the member's class. The same is true for non-public classes. The absence of `public`, `protected`, or `private` implies package-private.

You will often declare your class's instance fields to be `private` and provide special public instance methods for setting and getting their values. By convention, methods that set field values have names starting with `set` and are known as *setters*. Similarly, methods that get field values have names with `get` (or `is`, for Boolean fields) prefixes and are known as *getters*. Listing 3-25 demonstrates this pattern in the context of an `Employee` class declaration.

**Listing 3-25. Separation of Interface from Implementation**

```java
public class Employee {
    private String name;

    public Employee(String name) {
        setName(name);
    }

    public void setName(String empName) {
        name = empName; // Assign the empName argument to the name field.
    }

    public String getName() {
        return name;
    }
}
```

www.it-ebooks.info
Listing 3-25 presents an interface consisting of the public Employee class, its public constructor, and its public setter/getter methods. This class and these members can be accessed from anywhere. The implementation consists of the private name field and constructor/method code, which is only accessible within the Employee class.

It might seem pointless to go to all this bother when you could simply omit private and access the name field directly. However, suppose you’re told to introduce a new constructor that takes separate first and last name arguments and new methods that set/get the employee’s first and last names into this class. Furthermore, suppose that it’s been determined that the first and last names will be accessed more often than the entire name. Listing 3-26 reveals these changes.

Listing 3-26. Revising Implementation Without Affecting Existing Interface

```java
public class Employee {
    private String firstName;
    private String lastName;

    public Employee(String name) {
        setName(name);
    }

    public Employee(String firstName, String lastName) {
        setName(firstName + " " + lastName);
    }

    public void setName(String name) {
        // Assume that the first and last names are separated by a
        // single space character. indexOf() locates a character in a
        // string; substring() returns a portion of a string.
        setFirstName(name.substring(0, name.indexOf(' ')));
        setLastName(name.substring(name.indexOf(' ') + 1));
    }

    public String getName() {
        return getFirstName() + " " + getLastName();
    }

    public void setFirstName(String empFirstName) {
        firstName = empFirstName;
    }

    public String getFirstName() {
        return firstName;
    }
}
```
public void setLastName(String empLastName) {
    lastName = empLastName;
}

public String getLastName() {
    return lastName;
}
}

Listing 3-26 reveals that the name field has been removed in favor of new firstName and lastName fields, which were added to improve performance. Because setFirstName() and setLastName() will be called more frequently than setName(), and because getFirstName() and getLastName() will be called more frequently than getName(), it’s more performant (in each case) to have the first two methods set/get firstName’s and lastName’s values rather than merging either value into/extracting this value from name’s value.

Listing 3-26 also reveals setName() calling setFirstName() and setLastName(), and getName() calling getFirstName() and getLastName(), rather than directly accessing the firstName and lastName fields. Although avoiding direct access to these fields isn’t necessary in this example, imagine another implementation change that adds more code to setFirstName(), setLastName(), getFirstName(), and getLastName(); not calling these methods will result in the new code not executing.

Client code (code that instantiates and uses a class, such as Employee) will not break when Employee’s implementation changes from that shown in Listing 3-25 to that shown in Listing 3-26, because the original interface remains intact, although the interface has been extended. This lack of breakage results from hiding Listing 3-25’s implementation, especially the name field.

Note: setName() invokes the String class’s indexOf() and substring() methods. You’ll learn about these and other String methods in Chapter 7.

Java provides a little-known information hiding-related language feature that lets one object (or class method/initializer) access another object’s private fields or invoke its private methods. Listing 3-27 provides a demonstration.

Listing 3-27. One Object Accessing Another Object’s private Field

public class PrivateAccess {
    private int x;

    PrivateAccess(int x) {
        this.x = x;
    }
}
boolean equalTo(PrivateAccess pa) {
    return pa.x == x;
}

public static void main(String[] args) {
    PrivateAccess pa1 = new PrivateAccess(10);
    PrivateAccess pa2 = new PrivateAccess(20);
    PrivateAccess pa3 = new PrivateAccess(10);
    System.out.println("pa1 equal to pa2: " + pa1.equalTo(pa2));
    System.out.println("pa2 equal to pa3: " + pa2.equalTo(pa3));
    System.out.println("pa1 equal to pa3: " + pa1.equalTo(pa3));
    System.out.println(pa2.x);
}

Listing 3-27’s PrivateAccess class declares a private int field named x. It also declares an equalTo() method that takes a PrivateAccess argument. The idea is to compare the argument object with the current object to determine if they are equal.

The equality determination is made by using the == operator to compare the value of the argument object’s x instance field with the value of the current object’s x instance field, returning Boolean true when they are the same. What may seem baffling is that Java lets you specify pa.x to access the argument object's private instance field. Also, main() is able to access x directly, via the pa2 object.

I previously presented Java’s four access-control levels and the following statement about private access control: “A field, method, or constructor that is declared private cannot be accessed from beyond the class in which it’s declared.” When you carefully consider this statement and examine Listing 3-27, you’ll realize that x isn’t being accessed from beyond the PrivateAccess class in which it’s declared. Therefore, the private access-control level isn’t being violated.

The only code that can access this private instance field is code located within the PrivateAccess class. If you attempted to access x via a PrivateAccess object that was created in the context of another class, the compiler would report an error.

Being able to access x directly from within PrivateAccess is a performance enhancement; it’s faster to access this implementation detail directly than to call a method that returns its value.

Compile PrivateAccess.java as follows:

    javac PrivateAccess.java

Run the application as follows:

    java PrivateAccess
CHAPTER 3: Discovering Classes and Objects

You should observe the following output:

```java
pa1 equal to pa2: false
pa2 equal to pa3: false
pa1 equal to pa3: true
```

Tip  Get into the habit of developing useful interfaces while hiding implementations because it will save you a lot of trouble when maintaining your classes.

### Initializing Classes and Objects

Classes and objects need to be properly initialized before they are used. You’ve already learned that class fields are initialized to default zero values after a class loads and can be subsequently initialized by assigning values to them in their declarations via class field initializers, for example, `static int counter = 1;`. Similarly, instance fields are initialized to default values when an object’s memory is allocated via `new` and can be subsequently initialized by assigning values to them in their declarations via instance field initializers, for example, `int numDoors = 4;`.

Another aspect of initialization that’s already been discussed is the constructor, which is used to initialize an object, typically by assigning values to various instance fields, but is also capable of executing arbitrary code such as code that opens a file and reads the file’s contents.

Java provides two additional initialization features: class initializers and instance initializers. These features will be discussed in the following sections.

### Class Initializers

Constructors perform initialization tasks for objects. Their counterpart from a class initialization perspective is the class initializer.

A class initializer is a static-prefixed block that’s introduced into a class body. It’s used to initialize a loaded class via a sequence of statements. For example, I once used a class initializer to load a custom database driver class. Listing 3-28 shows the loading details.

```
Listing 3-28. Loading a Database Driver via a Class Initializer

class JDBCFilterDriver implements Driver
{
    static private Driver d;

    static
    {
        // Attempt to load JDBC-ODBC Bridge Driver and register that
        // driver.
    }
}
```
try
{
    Class c = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    d = (Driver) c.newInstance();
    DriverManager.registerDriver(new JDBCFilterDriver());
}
catch (Exception e)
{
    System.out.println(e);
}
//...

Listing 3-28’s JDBCFilterDriver class uses its class initializer to load and instantiate the class that describes Java’s JDBC-ODBC Bridge Driver and to register a JDBCFilterDriver instance with Java’s database driver. Although this listing’s JDBC-oriented code is probably meaningless to you right now, the listing illustrates the usefulness of class initializers. (I discuss JDBC in Chapter 14.)

A class can declare a mix of class initializers and class field initializers, as demonstrated in Listing 3-29.

**Listing 3-29. Mixing Class Initializers with Class Field Initializers**

class C {
    static {
        System.out.println("class initializer 1");
    }
    static int counter = 1;
    static {
        System.out.println("class initializer 2");
        System.out.println("counter = " + counter);
    }
}

Listing 3-29 declares a class named C that specifies two class initializers and one class field initializer. When the Java compiler compiles into a classfile a class that declares at least one class initializer or class field initializer, it creates a special void `<clinit>()` class method that stores the bytecode equivalent of all class initializers and class field initializers in the order they occur (from top to bottom).

---

**Note**  `<clinit>` isn’t a valid Java method name but is a valid name from the runtime perspective. The angle brackets were chosen as part of the name to prevent a name conflict with any `clinit()` methods that you might declare in the class.
For class C, <clinit>() would first contain the bytecode equivalent of System.out.println("class initializer 1"); it would next contain the bytecode equivalent of static int counter = 1;, and it would finally contain the bytecode equivalent of System.out.println("class initializer 2"); System.out.println("counter = " + counter);

When class C is loaded into memory, <clinit>() executes immediately and generates the following output:

class initializer 1
class initializer 2
counter = 1

**Instance Initializers**

Not all classes can have constructors, as you’ll discover in Chapter 5 when I present anonymous classes. For these classes, Java offers the instance initializer to handle instance initialization tasks.

An *instance initializer* is a block that’s introduced into a class body as opposed to being introduced as the body of a method or a constructor. The instance initializer is used to initialize an object via a sequence of statements, as demonstrated in Listing 3-30.

**Listing 3-30. Initializing a Pair of Arrays via an Instance Initializer**

```java
class Graphics {
    double[] sines;
    double[] cosines;

    {
        sines = new double[360];
        cosines = new double[sines.length];
        for (int degree = 0; degree < sines.length; degree++)
        {
            sines[degree] = Math.sin(Math.toRadians(degree));
            cosines[degree] = Math.cos(Math.toRadians(degree));
        }
    }
}
```

Listing 3-30’s Graphics class uses an instance initializer to create an object’s sines and cosines arrays and to initialize these arrays’ elements to the sines and cosines of angles ranging from 0 through 359 degrees. It does so because it’s faster to read array elements than to repeatedly call Math.sin() and Math.cos() elsewhere; performance matters. (In Chapter 7 I introduce Math.sin() and Math.cos().)

A class can declare a mix of instance initializers and instance field initializers, as shown in Listing 3-31.
Listing 3-31. Mixing Instance Initializers with Instance Field Initializers

class C
{
    {
        System.out.println("instance initializer 1");
    }

    int counter = 1;

    {
        System.out.println("instance initializer 2");
        System.out.println("counter = " + counter);
    }
}

Listing 3-31 declares a class named C that specifies two instance initializers and one instance field initializer. When the Java compiler compiles a class into a classfile, it creates a special void <init>() method representing the default noargument constructor when no constructor is explicitly declared; otherwise, it creates an <init>() method for each encountered constructor. Furthermore, it stores in each <init>() method the bytecode equivalent of all instance initializers and instance field initializers in the order in which they occur (from top to bottom), and before the constructor code.

Note <init> isn’t a valid Java method name, but it is a valid name from the runtime perspective. The angle brackets were chosen as part of the name to prevent a name conflict with any init() methods that you might declare in the class.

For class C, <init>() would first contain the bytecode equivalent of System.out.println("instance initializer 1");, it would next contain the bytecode equivalent of int counter = 1;, and it would finally contain the bytecode equivalent of System.out.println("instance initializer 2"); System.out.println("counter = " + counter);

When new C() executes, <init>() executes immediately and generates the following output:

| instance initializer 1        |
| instance initializer 2        |
| counter = 1                   |

Note You should rarely need to use the instance initializer, which isn’t commonly used in industry. Other developers would likely miss the instance initializer while scanning the source code and might find it confusing.
CHAPTER 3: Discovering Classes and Objects

Initialization Order

A class’s body can contain a mixture of class field initializers, class initializers, instance field initializers, instance initializers, and constructors. (You should prefer constructors to instance field initializers, although I’m guilty of not doing so consistently, and restrict your use of instance initializers to anonymous classes, discussed in Chapter 5.) Furthermore, class fields and instance fields initialize to default values. Understanding the order in which all of this initialization occurs is necessary to preventing confusion, so check out Listing 3-32.

Listing 3-32. A Complete Initialization Demo

```java
public class InitDemo {
    static double double1;
    double double2;
    static int int1;
    int int2;
    static String string1;
    String string2;

    static {
        System.out.println("[class] double1 = " + double1);
        System.out.println("[class] int1 = " + int1);
        System.out.println("[class] string1 = " + string1);
        System.out.println();
    }

    {
        System.out.println("[instance] double2 = " + double2);
        System.out.println("[instance] int2 = " + int2);
        System.out.println("[instance] string2 = " + string2);
        System.out.println();
    }

    static {
        double1 = 1.0;
        int1 = 1000000000;
        string1 = "abc";
    }

    {
        double2 = 1.0;
        int2 = 1000000000;
        string2 = "abc";
    }
}
```
Listing 3-32's InitDemo class declares two class fields and two instance fields for the double precision floating-point primitive type, one class field and one instance field for the integer primitive type, and one class field and one instance field for the String reference type. It also introduces one explicitly initialized class field, one explicitly initialized instance field, three class initializers,
three instance initializers, and one constructor. If you compile and run this code, you’ll observe the
following output:

```
[class] double1 = 0.0
[class] int1 = 0
[class] string1 = null

[class] double3 = 10.0

main() started

[class] double1 = 1.0
[class] double3 = 10.0
[class] int1 = 1000000000
[class] string1 = abc

About to create InitDemo object

[instance] double2 = 0.0
[instance] int2 = 0
[instance] string2 = null

[instance] double4 = 10.0

InitDemo() called

id created

[instance] id.double2 = 1.0
[instance] id.double4 = 10.0
[instance] id.int2 = 1000000000
[instance] id.string2 = abc

About to create InitDemo object

[instance] double2 = 0.0
[instance] int2 = 0
[instance] string2 = null

[instance] double4 = 10.0

InitDemo() called

id created

[instance] id.double2 = 1.0
[instance] id.double4 = 10.0
[instance] id.int2 = 1000000000
[instance] id.string2 = abc
```
As you study this output in conjunction with the aforementioned discussion of class initializers and instance initializers, you'll discover some interesting facts about initialization:

- Class fields initialize to default or explicit values just after a class is loaded. Immediately after a class loads, all class fields are zeroed to default values. Code within the `<clinit>()` method performs explicit initialization.
- All class initialization occurs prior to the `<clinit>()` method returning.
- Instance fields initialize to default or explicit values during object creation. When `new` allocates memory for an object, it zeros all instance fields to default values. Code within an `<init>()` method performs explicit initialization.
- All instance initialization occurs prior to the `<init>()` method returning.

Additionally, because initialization occurs in a top-down manner, attempting to access the contents of a class field before that field is declared or attempting to access the contents of an instance field before that field is declared causes the compiler to report an `illegal forward reference`.

**Collecting Garbage**

Objects are created via reserved word `new`, but how are they destroyed? Without some way to destroy objects, they will eventually fill up the heap’s available space and the application will not be able to continue. Java doesn’t provide the developer with the ability to remove them from memory. Instead, Java handles this task by providing a *garbage collector*, which is code that runs in the background and occasionally checks for unreferenced objects. When the garbage collector discovers an unreferenced object (or multiple objects that reference each other and where there are no other references to each other—only `A` references `B` and only `B` references `A`, for example), it removes the object from the heap, making more heap space available.

An *unreferenced object* is an object that cannot be accessed from anywhere within an application. For example, `new Employee("John", "Doe")` is an unreferenced object because the `Employee` reference returned by `new` is thrown away. In contrast, a *referenced object* is an object where the application stores at least one reference. For example, `Employee emp = new Employee("John", "Doe")`; is a referenced object because variable `emp` contains a reference to the `Employee` object.

A referenced object becomes unreferenced when the application removes its last stored reference. For example, if `emp` is a local variable that contains the only reference to an `Employee` object, this object becomes unreferenced when the method in which `emp` is declared returns. An application can also remove a stored reference by assigning `null` to its reference variable. For example, `emp = null` removes the reference to the `Employee` object that was previously stored in `emp`. 
Java’s garbage collector eliminates a form of memory leakage in C++ implementations that don’t rely on a garbage collector. In these C++ implementations, the developer must destroy dynamically-created objects before they go out of scope. If they vanish before destruction, they remain in the heap. Eventually, the heap fills and the application halts.

Although this form of memory leakage isn’t a problem in Java, a related form of leakage is problematic: continually creating objects and forgetting to remove even one reference to each object causes the heap to fill up and the application eventually to come to a halt. This form of memory leakage typically occurs in the context of collections (object-based data structures that store objects) and is a major problem for applications that run for lengthy periods of time; a web server is one example. For shorter-lived applications, you won’t normally notice this form of memory leakage. Consider Listing 3-33.

Listing 3-33. A Memory-Leaking Stack

```java
public class Stack
{
    private Object[] elements;
    private int top;

    public Stack(int size)
    {
        elements = new Object[size];
        top = -1; // indicate that stack is empty
    }

    public void push(Object o)
    {
        if (top + 1 == elements.length)
        {
            System.out.println("stack is full");
            return;
        }
        elements[++top] = o;
    }
}
```

Note The garbage collector scans an application’s object graph, which is a hierarchy of all of the objects currently stored in the heap. It starts with the current root set of references, which is a collection of local variables, parameters, class fields, and instance fields that currently exist and that contain (possibly null) references to objects. From this starting point, the garbage collector traces through subordinate references (references to objects stored in other objects, for example, a Car class’s string-based make field contains a reference to a String object) identifying all objects that are currently referenced. Objects that aren’t reachable via the root set of references are considered unreferenced and can be removed from the heap.
public Object pop()
{
    if (top == -1)
    {
        System.out.println("stack is empty");
        return null;
    }
    Object element = elements[top--];
    // elements[top + 1] = null;
    return element;
}

public static void main(String[] args)
{
    Stack stack = new Stack(2);
    stack.push("A");
    stack.push("B");
    stack.push("C");
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
}

Listing 3-33 describes a collection known as a stack, a data structure that stores elements in last-in, first-out order. Stacks are useful for remembering things, such as the instruction to return to when a method stops executing and must return to its caller.

Stack provides a push() method for pushing arbitrary objects onto the top of the stack and a pop() method for popping objects off of the stack's top in the reverse order to which they were pushed.

After creating a Stack object that can store a maximum of two objects, main() invokes push() three times, to push three String objects onto the stack. Because the stack's internal array can store two objects only, push() outputs an error message when main() tries to push "C".

At this point, main() attempts to pop three Objects off of the stack, outputting each object to the standard output device. The first two pop() method calls succeed, but the final method call fails and outputs an error message because the stack is empty when it's called.

When you run this application, it generates the following output:

stack is full
B
A
stack is empty
null

There is a problem with the Stack class: it leaks memory. When you push an object onto the stack, its reference is stored in the internal elements array. When you pop an object off of the stack, the object's reference is obtained and top is decremented, but the reference remains in the array (until you invoke push()).
Imagine a scenario where the Stack object's reference is assigned to a class field, which means that the Stack object hangs around for the life of the application. Furthermore, suppose that you have pushed three 50-megabyte Image objects onto the stack and then subsequently popped them off of the stack. After using these objects, you assign null to their reference variables, thinking that they will be garbage collected the next time the garbage collector runs. However, this won’t happen because the Stack object still maintains its references to these objects, and so 150 megabytes of heap space won’t be available to the application, and maybe the application will run out of memory.

The solution to this problem is for pop() to explicitly assign null to the elements entry before returning the reference. Simply uncomment the elements[top + 1] = null; line in Listing 3-33 to make this happen.

You might think that you should always assign null to reference variables when their referenced objects are no longer required. However, doing so often doesn’t improve performance or free up significant amounts of heap space and can lead to thrown instances of the java.lang.NullPointerException class when you’re not careful. (I discuss NullPointerException in the context of Chapter 5’s coverage of Java’s exceptions-oriented language features). You typically nullify reference variables in classes that manage their own memory, such as the aforementioned Stack class.

---

**Note**  To learn more about garbage collection in a Java 5 context, check out Oracle’s “Memory Management in the Java HotSpot Virtual Machine” whitepaper at [www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf](http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf)

---

### Revisiting Arrays

In Chapter 2, I introduced you to arrays, which are regions of memory (specifically, the heap) that store values in equal-size and contiguous slots, known as elements. I also presented several examples, including the following:

```java
char gradeLetters[] = { 'A', 'B', 'C', 'D', 'F' };
```

Here you have an array variable named gradeLetters that stores a reference to a five-element region of memory, which stores the characters A, B, C, D, and F in contiguous and equal-size (16-bit) memory locations.

---

**Note**  I’ve placed the [] brackets after gradeLetters. Although this is legal, it’s conventional to place these brackets after the type name, as in char[] gradeLetters = { 'A', 'B', 'C', 'D', 'F' };
I demonstrate both approaches in this section.

---

You access an element by specifying gradeLetters[x], where x is an integer that identifies an array element and is known as an index; the first array element is always located at index 0. The following example shows you how to output and change the first element’s value:

```java
System.out.println(gradeLetters[0]); // Output the first grade letter.
gradeLetters[0] = 'a';               // Perhaps you prefer lowercase grade letters.
```
CHAPTER 3: Discovering Classes and Objects

The { 'A', 'B', 'C', 'D', 'F' } array-creation syntax is an example of syntactic sugar (syntax that simplifies a language, making it “sweeter” to use). Behind the scenes, the array is created with the new operator and initializes to these values, as follows:

```java
char gradeLetters[] = new char[] { 'A', 'B', 'C', 'D', 'F' };
```

First, a five-character region of memory is allocated. Next, the region’s five character elements are initialized to A, B, C, D, and F. Finally, a reference to these elements is stored in array variable gradeLetters.

**Caution** It’s an error to place an integer value between the square brackets following char. For example, the compiler reports an error when it encounters the 5 in

```java
```

You can think of an array as a special kind of object, although it’s not an object in the same sense that a class instance is an object. This pseudo-object has a solitary and read-only length field that contains the array’s size (the number of elements). For example, gradeLetters.length returns the number of elements (5) in the gradeLetters array.

Although you can use either of the previous two approaches to create an array, you will often specify a third approach that doesn’t involve explicit element initialization and subsequently initialize the array. This approach is demonstrated by the following code:

```java
char gradeLetters[] = new char[5];
```

You specify the number of elements as a positive integer between the square brackets. Operator new zeros the bits in each array element’s storage location, which you interpret at the source code level as literal value false, \u0000, 0, 0L, 0.0, 0.0F, or null (depending on element type).

You can then initialize the array, as follows:

```java
gradeLetters[0] = 'A';
gradeLetters[1] = 'B';
gradeLetters[2] = 'C';
gradeLetters[3] = 'D';
gradeLetters[4] = 'F';
```

However, you will probably find it more convenient to use a loop for this task, as follows:

```java
for (int i = 0; i < gradeLetters.length; i++)
    gradeLetters[i] = 'A' + i;
```

The previous examples focused on creating an array whose values share a common primitive type (character, represented by the char keyword). You can also create an array of object references. For example, you can create an array to store three Image object references, as follows:

```java
Image[] imArray = { new Image("image0.png"), new Image("image1.png"), new Image("image2.png") };
```
Here you have an array variable named `imArray` that stores a reference to a three-element region of memory, where each element stores a reference to an `Image` object. The `Image` object is located elsewhere in memory.

You access an `Image` element by specifying `imArray[x]`. The following example assumes the existence of a `getLength()` method that returns the image's length (in bytes) and calls this method on the first `Image` object to return the first image's length, which is subsequently output:

```java
System.out.println(imArray[0].getLength());
```

As with the previous `gradeLetters` example, you can combine the `new` operator with the syntactic sugar initializer, as follows:

```java
Image[] imArray = new Image[] { new Image("image0.png"), new Image("image1.png"),
                                new Image("image2.png") };
```

Finally, you can use the third approach, which initializes each object reference to the null reference by setting all of the bits in each element to 0. This approach is demonstrated as follows:

```java
Image[] imArray = new Image[3];
```

Because `new` initializes each element to the null reference, you must explicitly initialize this array, and you can conveniently do so as follows:

```java
for (int i = 0; i < imArray.length; i++)
    imArray[i] = new Image("image" + i + ".png"); // image0.png, image1.png, and so on
```

The "image" + i + ".png" expression uses the string concatenation operator (+) to combine image with the string equivalent of the integer value stored in variable i with .png. The resulting string is passed to `Image`'s `Image(String filename)` constructor, and the resulting reference is stored in one of the array elements.

**Note** Use of the string concatenation operator in a loop context can result in a lot of unnecessary `String` object creation, depending on the length of the loop. I will discuss this topic in Chapter 7 when I introduce you to the `String` class.

The previous examples have focused on creating one-dimensional arrays. However, you can also create multidimensional arrays (that is, arrays with two or more dimensions). For example, consider a two-dimensional array of temperature values.

Although you can use any of the three approaches to create the `temperatures` array, the third approach is preferable when the values vary greatly. The following example creates this array as a three-row-by-two-column table of double precision floating-point temperature values:

```java
double[][] temperatures = new double[3][2];
```

Notice the two sets of square brackets between `double` and `temperatures`. These two sets of brackets signify the array as two-dimensional (a table). Also notice the two sets of square brackets following `new` and `double`. Each set contains a positive integer value signifying the number of rows (3) or the number of columns (2) for each row.
After creating the array, you can populate its elements with suitable values. The following example initializes each temperatures element, which is accessed as temperatures[row][col], to a randomly generated temperature value via Math.random(), which I’ll explain in Chapter 7:

```java
for (int row = 0; row < temperatures.length; row++)
    for (int col = 0; col < temperatures[row].length; col++)
        temperatures[row][col] = Math.random() * 100;
```

The outer for loop selects each row from row 0 to the length of the array (which identifies the number of rows in the array). The inner for loop selects each column from 0 to the length of the current row array (which identifies the number of columns represented by that array). In essence, you’re looking at a one-dimensional row array where each element references a one-dimensional column array.

You can subsequently output these values in a tabular format by using another for loop, as demonstrated by the following example where the code makes no attempt to align the temperature values in perfect columns:

```java
for (int row = 0; row < temperatures.length; row++)
{
    for (int col = 0; col < temperatures[row].length; col++)
        System.out.print(temperatures[row][col] + " ");
    System.out.println();
}
```

Java provides an alternative for creating a multidimensional array in which you create each dimension separately. For example, to create the previous two-dimensional temperatures array via new in this manner, first create a one-dimensional row array (the outer array), and then create a one-dimensional column array (the inner array), like so:

```java
// Create the row array.
double[][][] temperatures = new double[3][]; // Note the extra empty pair of brackets.
// Create a column array for each row.
for (int row = 0; row < temperatures.length; row++)
    temperatures[row] = new double[2]; // 2 columns per row
```

When you specify a different number of columns for each row, the resulting array is known as a **ragged array** because the array isn’t rectangular.

```
Note When creating the row array, you must specify an extra pair of empty brackets as part of the expression following new. (For a three-dimensional array—a one-dimensional array of tables, where this array’s elements reference row arrays—you must specify two pairs of empty brackets as part of the expression following new.)
```
The following exercises are designed to test your understanding of Chapter 3’s content.

1. What is a class?
2. How do you declare a class?
3. True or false: You can declare multiple public classes in a source file.
4. What is an object?
5. How do you obtain an object?
6. What is a constructor?
7. True or false: Java creates a default no-argument constructor when a class declares no constructors.
8. What is a parameter list and what is a parameter?
9. What is an argument list and what is an argument?
10. True or false: You invoke another constructor by specifying the name of the class followed by an argument list.
12. What is a local variable?
15. What is encapsulation?
16. Define field.
17. What is the difference between an instance field and a class field?
18. What is a blank final, and how does it differ from a true constant?
19. How do you prevent a field from being shadowed?
21. What is the difference between an instance method and a class method?
22. Define recursion.
23. How do you overload a method?
24. What is a class initializer and an instance initializer?
25. Define garbage collector.
26. What is an object graph?
28. What is a ragged array?
29. Merge Listings 3-6 and 3-7 into a complete Image application that demonstrates its constructors.
30. Create a `Conversions` class with `c2f()` and `f2c()` class methods that convert their double arguments to degrees Fahrenheit or degrees Celsius. Introduce a `main()` method into this class to test these methods.

31. Create a `Utilities` class that incorporates the previous `factorial()` methods (using iteration and recursion) along with the `sum()` method that used the variable arguments feature to sum a variable number of arguments. Modify the recursive `factorial()` method to also handle the case of 0!, which equals 1. Introduce a `main()` method into this class that demonstrates these methods.

32. The `factorial()` method provides an example of tail recursion, a special case of recursion in which the method’s last statement contains a recursive call, which is known as a tail call. Provide another example of tail recursion in which you create a GCD application whose static `int gcd(int a, int b)` class method returns the highest integer that divides evenly into both arguments. In other words, this method returns the greatest common divisor of the integer arguments passed to `a` and `b`.

33. Create a `Book` class with private name, author, and International Standard Book Number (ISBN) fields. Provide a suitable constructor and getter methods that return field values. Introduce a `main()` method into this class that creates an array of `Book` objects and iterates over this array outputting each book’s name, author, and ISBN.

### Summary

A class is a container for housing an application, and it is also a template for manufacturing objects. You declare a class by minimally specifying reserved word `class` followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code), followed by a body. The body starts with an open brace character (`{`) and ends with a close brace (`}`). Sandwiched between these delimiters are various kinds of member declarations.

Objects are instances of classes. You create objects by using the `new` operator to allocate memory and a constructor to initialize the object. A constructor is a block of code that’s declared in a class for constructing an object from that class by initializing it in some manner. The `new` operator returns a reference to the newly created and initialized object.

A constructor doesn’t have a name. Instead, you must specify the name of the class that declares the constructor. This name is followed by a round bracket-delimited parameter list, which is a comma-separated list of zero or more parameter declarations. A parameter is a constructor or method variable that receives an expression value passed to the constructor or method when it’s called. This expression value is known as an argument.

When a class doesn’t declare a constructor, Java implicitly creates a constructor for that class. The created constructor is known as the default noargument constructor because no arguments appear between its `( ` and `)` characters when the constructor is invoked. The default noargument constructor isn’t created when at least one constructor is declared in the class.

Classes typically combine state with behaviors. State refers to attributes that are read and/or written when an application runs, and behaviors refer to sequences of code that read/write attributes and perform other tasks. Combining state with behaviors is known as encapsulation.

Java represents state via fields, which are variables declared within a class’s body. State associated with a class is described by class fields, whereas state associated with objects is described by object fields (also known as instance fields).
Java represents behaviors via methods, which are named blocks of code declared within a class’s body. Behaviors associated with a class are described by class methods, whereas behaviors associated with objects are described by object methods (also known as instance methods).

Every class exposes an interface, which are the constructors, methods, and (possibly) fields that can be accessed from outside of the class. An interface serves as a contract between a class and its clients, which are external classes that communicate with the class and/or its instances by accessing fields and calling constructors and methods. The contract is such that the class promises to not change its interface, which would break dependent clients.

The class also provides an implementation (the code within exposed methods along with optional helper methods and optional supporting fields that shouldn’t be exposed) that codifies the interface. Helper methods assist exposed methods and shouldn’t be exposed.

When designing a class, your goal is to expose a useful interface while hiding details of that interface’s implementation. You hide the implementation to prevent developers from accidentally accessing parts of your class that don’t belong to the class’s interface so that you’re free to change the implementation without breaking client code. Hiding the implementation is often referred to as information hiding. Furthermore, many developers consider implementation hiding to be part of encapsulation.

Java supports implementation hiding by providing four levels of access control, where three of these levels are indicated via a reserved word: public, private, and protected. You can use these access control levels to control access to fields, methods, and constructors and two of these levels to control access to classes.

Classes and objects need to be properly initialized before they’re used. You’ve already learned that class fields are initialized to default zero values after a class loads and can be subsequently initialized by assigning values to them in their declarations via class field initializers. Similarly, instance fields are initialized to default values when an object’s memory is allocated via new and can be subsequently initialized by assigning values to them in their declarations via instance field initializers or via constructors.

Java also supports class initializers and instance initializers for this task. A class initializer is a static-prefixed block that’s introduced into a class body. It’s used to initialize a loaded class via a sequence of statements. An instance initializer is a block that’s introduced into a class body as opposed to being introduced as the body of a method or a constructor. The instance initializer is used to initialize an object via a sequence of statements.

Objects are created via reserved word new, but how are they destroyed? Without some way to destroy objects, they will eventually fill up the heap’s available space and the application will not be able to continue. Java doesn’t provide the developer with the ability to remove them from memory. Instead, Java handles this task by providing a garbage collector, which is code that runs in the background and occasionally checks for unreferenced objects.

You can think of an array as a special kind of object, although it’s not an object in the same sense that a class instance is an object. This pseudo-object has a solitary and read-only length field that contains the array’s size (the number of elements). In addition to using the syntactic sugar first presented in Chapter 2 for creating an array, you can also create an array using the new operator, with or without the syntactic sugar.

Chapter 4 continues to explore the Java language by examining its support for inheritance, polymorphism, and interfaces.
Discovering Inheritance, Polymorphism, and Interfaces

An object-based language is a language that encapsulates state and behaviors in objects. Java’s support for encapsulation (discussed in Chapter 3) qualifies it as an object-based language. However, Java is also an object-oriented language because it supports inheritance and polymorphism (as well as encapsulation). (Object-oriented languages are a subset of object-based languages.) In this chapter, I will introduce you to Java’s language features that support inheritance and polymorphism. Also, I will introduce you to interfaces, Java’s ultimate abstract type mechanism.

Building Class Hierarchies

We tend to categorize stuff by saying things like “cars are vehicles” or “savings accounts are bank accounts.” By making these statements, we really are saying (from a software development perspective) that cars inherit vehicular state (such as make and color) and behaviors (such as park and display mileage) and that savings accounts inherit bank account state (such as balance) and behaviors (such as deposit and withdraw). Car, vehicle, savings account, and bank account are examples of real-world entity categories, and inheritance is a hierarchical relationship between similar entity categories in which one category inherits state and behaviors from at least one other entity category. Inheriting from a single category is single inheritance, and inheriting from at least two categories is multiple inheritance.

Java supports single inheritance in a class context to facilitate code reuse—why reinvent the wheel? In this context, a class inherits state and behaviors from another class through class extension. Because classes are involved, Java refers to this kind of inheritance as implementation inheritance.

Java supports multiple inheritance and also supports single inheritance in an interface context in which a class inherits behavior templates from one or more interfaces through interface implementation or in which an interface inherits behavior templates from one or more interfaces.
through interface extension. Because interfaces are involved, Java refers to this kind of inheritance as *interface inheritance*. (I discuss interfaces later in this chapter.)

**Note** You reuse code by carefully extending classes, implementing interfaces, and extending interfaces. You start with something that is close to what you want and then you extend it to meet your goal. You don’t reuse code by simply copying and pasting it. Copying and pasting often results in redundant (i.e., non-reusable) and buggy code.

In the next section, I will introduce you to Java’s support for implementation inheritance by first focusing on class extension. I will then introduce you to a special class that sits at the top of Java’s class hierarchy. After introducing you to composition, which is an alternative to implementation inheritance for reusing code, I will show you how composition can be used to overcome problems with implementation inheritance.

## Extending Classes
Java provides the reserved word `extends` for specifying a hierarchical relationship between two classes. For example, suppose you have a *Vehicle* class and want to introduce a *Car* class as a kind of *Vehicle*. Listing 4-1 uses `extends` to cement this relationship.

**Listing 4-1. Relating Two Classes via `extends`**

```java
class Vehicle
{
    // member declarations
}

class Car extends Vehicle
{
    // member declarations
}
```

Listing 4-1 codifies a relationship that is known as an “is-a” relationship: a car is a kind of vehicle. In this relationship, *Vehicle* is known as the *base class*, *parent class*, or *superclass*; and *Car* is known as the *derived class*, *child class*, or *subclass*.

**Note** When you don’t want anyone to extend one of your classes (for security or another reason), you must declare that class `final`. For example, if you specified `final class Vehicle {}`, the compiler would report an error upon encountering `class Car extends Vehicle {}`. 
As well as being capable of providing its own member declarations, Car is capable of inheriting member declarations from its Vehicle superclass. As Listing 4-2 shows, non-private inherited members become accessible to members of the Car class.

Listing 4-2. Inheriting Members

class Vehicle
{
    private String make;
    private String model;
    private int year;

    Vehicle(String make, String model, int year)
    {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    String getMake()
    {
        return make;
    }

    String getModel()
    {
        return model;
    }

    int getYear()
    {
        return year;
    }
}

public class Car extends Vehicle
{
    private int numWheels;

    Car(String make, String model, int year, int numWheels)
    {
        super(make, model, year);
        this.numWheels = numWheels;
    }

    public static void main(String[] args)
    {
        Car car = new Car("Ford", "Fiesta", 2009, 4);
        System.out.println("Make = " + car.getMake());
        System.out.println("Model = " + car.getModel());
        System.out.println("Year = " + car.getYear());
        // Normally, you cannot access a private field via an object
A class whose instances cannot be modified is known as an immutable class. Vehicle is an example. If Car's main() method, which can directly read or write numWheels, was not present, Car would also be an example of an immutable class. Also, a class cannot inherit constructors, nor can it inherit private fields and methods. For example, Car doesn’t inherit Vehicle’s constructor, nor does it inherit Vehicle’s private make, model, and year fields.
A subclass can override (replace) an inherited method so that the subclass’s version of the method is called instead. Listing 4-3 shows you that the overriding method must specify the same name, parameter list, and return type as the method being overridden.

Listing 4-3. Overriding a Method

class Vehicle
{
    private String make;
    private String model;
    private int year;

    Vehicle(String make, String model, int year)
    {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    void describe()
    {
        System.out.println(year + " " + make + " " + model);
    }
}

public class Car extends Vehicle
{
    private int numWheels;

    Car(String make, String model, int year, int numWheels)
    {
        super(make, model, year);
    }

    void describe()
    {
        System.out.print("This car is a "); // Print without newline – see Chapter 1.
        super.describe();
    }

    public static void main(String[] args)
    {
        Car car = new Car("Ford", "Fiesta", 2009, 4);
        car.describe();
    }
}

Listing 4-3’s Car class declares a describe() method that overrides Vehicle’s describe() method to output a car-oriented description. After outputting an initial message, this method uses reserved word super to call Vehicle’s describe() method via super.describe();.
CHAPTER 4: Discovering Inheritance, Polymorphism, and Interfaces

Note Call a superclass method from the overriding subclass method by prefixing the method's name with reserved word super and the member access operator. If you don’t do this, you end up recursively calling the subclass’s overriding method. Use super and the member access operator to access non-private superclass fields from subclasses that mask these fields by declaring same-named fields.

If you were to compile Listing 4-3 (javac Car.java) and run the Car application (java Car), you would discover that Car’s overriding describe() method executes instead of Vehicle’s overridden describe() method and outputs This car is a 2009 Ford Fiesta.

Note When you don’t want anyone to extend one of your methods (for security or another reason), you must declare that method final. For example, if you specified final void describe() for Vehicle’s describe() method, the compiler would report an error upon encountering an attempt to override this method in the Car class. Also, you cannot make an overriding method less accessible than the method it overrides. For example, if Car’s describe() method was declared as private void describe(), the compiler would report an error because private access is less accessible than the default package access. However, describe() could be made more accessible by declaring it public, as in public void describe().

Suppose you happened to replace Listing 4-3’s describe() method with the method shown here:

```java
void describe(String owner) {
    System.out.print("This car, which is owned by " + owner + ", is a ");
    super.describe();
}
```

The modified Car class now has two describe() methods, the preceding explicitly declared method and the method inherited from Vehicle. The void describe(String owner) method doesn’t override Vehicle’s describe() method. Instead, it overloads this method.

The Java compiler helps you detect an attempt to overload instead of override a method at compile time by letting you prefix a subclass’s method header with the @Override annotation, as shown below. (I will discuss annotations in Chapter 6.)

```java
@Override
void describe() {
    System.out.print("This car is a ");
    super.describe();
}
```
Specifying `@Override` tells the compiler that the method overrides another method. If you overload the method instead, the compiler reports an error. Without this annotation, the compiler would not report an error because method overloading is a valid feature.

*Tip* Get into the habit of prefixing overriding methods with the `@Override` annotation. This habit will help you detect overloading mistakes much sooner.

In Chapter 3, I discussed the initialization order of classes and objects, where you learned that class members are always initialized first and in a top-down order (the same order applies to instance members). Implementation inheritance adds a couple more details:

- A superclass’s class initializers always execute before a subclass’s class initializers.
- A subclass’s constructor always calls the superclass constructor to initialize an object’s superclass layer and then initializes the subclass layer.

Java’s support for implementation inheritance only permits you to extend a single class. You cannot extend multiple classes because doing so can lead to problems. For example, suppose Java supported multiple implementation inheritance, and you decided to model a flying horse (from Greek mythology) via the class structure shown in Listing 4-4.

**Listing 4-4. A Fictional Demonstration of Multiple Implementation Inheritance**

```java
class Bird
{
    void describe()
    {
        // code that outputs a description of a bird's appearance and behaviors
    }
}

class Horse
{
    void describe()
    {
        // code that outputs a description of a horse's appearance and behaviors
    }
}

class FlyingHorse extends Bird, Horse
{
    public static void main(String[] args)
    {
        FlyingHorse pegasus = new FlyingHorse();
        pegasus.describe();
    }
}
```
Listing 4-4’s class structure reveals an ambiguity resulting from each of Bird and Horse declaring a describe() method. Which of these methods does FlyingHorse inherit? A related ambiguity arises from same-named fields, possibly of different types. Which field is inherited?

**The Ultimate Superclass**

A class that doesn’t explicitly extend another class implicitly extends Java’s Object class (located in the java.lang package—I will discuss packages in the next chapter). For example, Listing 4-1’s Vehicle class extends Object, whereas Car extends Vehicle.

Object is Java’s ultimate superclass because it serves as the ancestor of every other class but doesn’t itself extend any other class. Object provides a common set of methods that other classes inherit. Table 4-1 describes these methods.

**Table 4-1. Object’s Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object clone()</td>
<td>Create and return a copy of the current object.</td>
</tr>
<tr>
<td>boolean equals(Object obj)</td>
<td>Determine if the current object is equal to the object identified by obj.</td>
</tr>
<tr>
<td>void finalize()</td>
<td>Finalize the current object.</td>
</tr>
<tr>
<td>Class&lt;?&gt; getClass()</td>
<td>Return the current object’s java.lang.Class object.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Return the current object’s hash code.</td>
</tr>
<tr>
<td>void notify()</td>
<td>Wake up one of the threads that are waiting on the current object’s monitor.</td>
</tr>
<tr>
<td>void notifyAll()</td>
<td>Wake up all threads that are waiting on the current object’s monitor.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Return a string representation of the current object.</td>
</tr>
<tr>
<td>void wait()</td>
<td>Cause the current thread to wait on the current object’s monitor until it is woken up via notify() or notifyAll().</td>
</tr>
<tr>
<td>void wait(long timeout)</td>
<td>Cause the current thread to wait on the current object’s monitor until it is woken up via notify() or notifyAll() or until the specified timeout value (in milliseconds) has elapsed, whichever comes first.</td>
</tr>
<tr>
<td>void wait(long timeout, int nanos)</td>
<td>Cause the current thread to wait on the current object’s monitor until it is woken up via notify() or notifyAll() or until the specified timeout value (in milliseconds) plus nanos value (in nanoseconds) has elapsed, whichever comes first.</td>
</tr>
</tbody>
</table>

I will discuss the clone(), equals(), finalize(), hashCode(), and toString() methods shortly, but will defer a discussion of notify(), notifyAll(), and the wait() methods until Chapter 7, and defer a discussion of the getClass() method until Chapter 8.
Cloning

The clone() method clones (duplicates) an object without calling a constructor. It copies each primitive or reference field's value to its counterpart in the clone, a task known as shallow copying or shallow cloning. Listing 4-5 demonstrates this behavior.

Listing 4-5. Shallowly Cloning an Employee Object

```java
public class Employee implements Cloneable {
    String name;
    int age;

    Employee(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Employee e1 = new Employee("John Doe", 46);
        Employee e2 = (Employee) e1.clone();
        System.out.println(e1 == e2); // Output: false
        System.out.println(e1.name == e2.name); // Output: true
    }
}
```

Listing 4-5 declares an Employee class with name and age instance fields and a constructor for initializing these fields. The main() method uses this constructor to initialize a new Employee object's copies of these fields to John Doe and 46.

Note: A class must implement the java.lang.Cloneable interface or its instances cannot be shallowly cloned via Object's clone() method—this method performs a runtime check to see if the class implements Cloneable. (I will discuss interfaces later in this chapter.) If a class doesn't implement Cloneable, clone() throws java.lang.CloneNotSupportedException. (Because CloneNotSupportedException is a checked exception, it's necessary for Listing 4-5 to satisfy the compiler by appending throws CloneNotSupportedException to the main() method's header. I will discuss exceptions in the next chapter.) The java.lang.String class is an example of a class that doesn't implement Cloneable; hence, String objects cannot be shallowly cloned.

After assigning the Employee object's reference to local variable e1, main() calls the clone() method on this variable to duplicate the object and then assigns the resulting reference to variable e2. The (Employee) cast is needed because clone() returns Object.
To prove that the objects whose references were assigned to e1 and e2 are different, main() next compares these references via == and outputs the Boolean result, which happens to be false. To prove that the Employee object was shallowly cloned, main() next compares the references in both Employee objects’ name fields via == and outputs the Boolean result, which happens to be true.

**Note** In Listing 4-5, I didn’t override Object’s clone() method because the e1.clone() method call occurred in the class whose instances were to be cloned. To clone an Employee object from another class (such as UseEmployee), I would have to override clone(), as follows: @Override public Object clone() throws CloneNotSupportedException { return super.clone(); }.

Shallow cloning is not always desirable because the original object and its clone refer to the same object via their equivalent reference fields. For example, each of Listing 4-5’s two Employee objects refers to the same String object via its name field.

Although not a problem for String, whose instances are immutable, changing a mutable object via the clone’s reference field causes the original (noncloned) object to see the same change via its reference field. For example, suppose you add a reference field named hireDate to Employee. This field is of type Date with year, month, and day instance fields. Because Date is intended to be mutable, you can change the contents of these fields in the Date instance assigned to hireDate.

Now suppose you plan to change the clone’s date, but want to preserve the original Employee object’s date. You cannot do this with shallow cloning because the change is also visible to the original Employee object. To solve this problem, you must modify the cloning operation so that it assigns a new Date reference to the Employee clone’s hireDate field. This task, which is known as deep copying or deep cloning, is demonstrated in Listing 4-6.

**Listing 4-6. Deeply Cloning an Employee Object**

class Date
{
    int year, month, day;

    Date(int year, int month, int day)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }
}

public class Employee implements Cloneable
{
    String name;
    int age;
    Date hireDate;
Employee(String name, int age, Date hireDate) {
    this.name = name;
    this.age = age;
    this.hireDate = hireDate;
}

@Override
protected Object clone() throws CloneNotSupportedException {
    Employee emp = (Employee) super.clone();
    if (hireDate != null) // no point cloning a null object (one that doesn't exist)
        emp.hireDate = new Date(hireDate.year, hireDate.month, hireDate.day);
    return emp;
}

public static void main(String[] args) throws CloneNotSupportedException {
    Employee e1 = new Employee("John Doe", 46, new Date(2000, 1, 20));
    Employee e2 = (Employee) e1.clone();
    System.out.println(e1 == e2); // Output: false
    System.out.println(e1.name == e2.name); // Output: true
    System.out.println(e1.hireDate == e2.hireDate); // Output: false
    System.out.println(e2.hireDate.year + " " + e2.hireDate.month + " " +
                        e2.hireDate.day); // Output: 2000 1 20
}

Listing 4-6 declares Date and Employee classes. The Date class declares year, month, and day fields and a constructor. (You can declare a comma-separated list of variables on one line provided that these variables all share the same type, which is int in this case.)

Employee overrides the clone() method to deeply clone the hireDate field. This method first calls Object's clone() method to shallowly clone the current Employee object's instance fields and then stores the new object's reference in emp. Assuming that hireDate doesn't contain the null reference, it next assigns a new Date object's reference to emp's hireDate field; this object's fields are initialized to the same values as those in the original Employee object's hireDate instance.

At this point, you have an Employee clone with shallowly cloned name and age fields and a deeply cloned hireDate field. The clone() method finishes by returning this Employee clone.

Note: If you're not calling Object's clone() method from an overriding clone() method (because you prefer to deeply clone reference fields and do your own shallow copying of nonreference fields), it isn't necessary for the class containing the overriding clone() method to implement Cloneable, but it should implement this interface for consistency. String doesn't override clone(), so String objects cannot be deeply cloned.
Equality

The == and != operators compare two primitive values (such as integers) for equality (==) or inequality (!=). These operators also compare two references to see whether they refer to the same object or not. This latter comparison is known as an identity check.

You cannot use == and != to determine whether two objects are logically the same (or not). For example, two Car objects with the same field values are logically equivalent. However, == reports them as unequal because of their different references.

Note Because == and != perform the fastest possible comparisons and because string comparisons need to be performed quickly (especially when sorting a huge number of strings), the String class contains special support that allows literal strings and string-valued constant expressions to be compared via == and !=. (I will discuss this support when I present String in Chapter 7.) The following statements demonstrate these comparisons:

```java
System.out.println("abc" == "abc"); // Output: true
System.out.println("abc" == "a" + "bc"); // Output: true
System.out.println("abc" == "Abc"); // Output: false
System.out.println("abc" != "def"); // Output: true
System.out.println("abc" == new String("abc")); // Output: false
```

Recognizing the need to support logical equality in addition to reference equality, Java provides an `equals()` method in the `Object` class. Because this method defaults to comparing references, you need to override `equals()` to compare object contents.

Before overriding `equals()`, make sure that this is necessary. For example, Java's `java.lang.StringBuffer` class doesn't override `equals()`. Perhaps this class's designers didn’t think it necessary to determine whether two StringBuffer objects are logically equivalent or not.

You cannot override `equals()` with arbitrary code. Doing so will probably prove disastrous to your applications. Instead, you need to adhere to the contract that is specified in the Java documentation for this method, which I present next.

The `equals()` method implements an equivalence relation on nonnull object references:

- **It is reflexive**: For any nonnull reference value `x`, `x.equals(x)` returns true.
- **It is symmetric**: For any nonnull reference values `x` and `y`, `x.equals(y)` returns true if and only if `y.equals(x)` returns true.
It is transitive: For any nonnull reference values \( x, y, \) and \( z, \) if \( x.equals(y) \) returns true and \( y.equals(z) \) returns true, then \( x.equals(z) \) returns true.

It is consistent: For any nonnull reference values \( x \) and \( y, \) multiple invocations of \( x.equals(y) \) consistently return true or consistently return false, provided no information used in \( equals() \) comparisons on the objects is modified.

For any nonnull reference value \( x, \) \( x.equals(null) \) returns false.

Although this contract probably looks somewhat intimidating, it isn’t that difficult to satisfy. For proof, take a look at the implementation of the \( equals() \) method in Listing 4-7’s \( \text{Point} \) class.

Listing 4-7. Logically Comparing \( \text{Point} \) Objects

```java
public class Point
{
    private int x, y;

    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    int getX()
    {
        return x;
    }

    int getY()
    {
        return y;
    }

    @Override
    public boolean equals(Object o)
    {
        if (!(o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }

    public static void main(String[] args)
    {
        Point p1 = new Point(10, 20);
        Point p2 = new Point(20, 30);
        Point p3 = new Point(10, 20);
        // Test reflexivity
        System.out.println(p1.equals(p1)); // Output: true
        // Test symmetry
        System.out.println(p1.equals(p2)); // Output: false
        System.out.println(p2.equals(p1)); // Output: false
    }
}
```

www.it-ebooks.info
CHAPTER 4: Discovering Inheritance, Polymorphism, and Interfaces

// Test transitivity
System.out.println(p2.equals(p3)); // Output: false
System.out.println(p1.equals(p3)); // Output: true
// Test nullability
System.out.println(p1.equals(null)); // Output: false
// Extra test to further prove the instanceof operator's usefulness.
System.out.println(p1.equals("abc")); // Output: false

Listing 4-7's overriding equals() method begins with an if statement that uses the instanceof operator to determine whether the argument passed to parameter o is an instance of the Point class. If not, the if statement executes return false;

The o instanceof Point expression satisfies the last portion of the contract: for any nonnull reference value x, x.equals(null) returns false. Because the null reference is not an instance of any class, passing this value to equals() causes the expression to evaluate to false.

The o instanceof Point expression also prevents a java.lang.ClassCastException instance from being thrown via expression (Point) o in the event that you pass an object other than a Point object to equals(). (I will discuss exceptions in the next chapter.)

Following the cast, the contract's reflexivity, symmetry, and transitivity requirements are met by only allowing Points to be compared with other Points via expression p.x == x && p.y == y.

The final contract requirement, consistency, is met by making sure that the equals() method is deterministic. In other words, this method doesn't rely on any field value that could change from method call to method call.

Tip You can optimize the performance of a time-consuming equals() method by first using == to determine if o's reference identifies the current object. Simply specify if (o == this) return true; as the equals() method's first statement. This optimization isn't necessary in Listing 4-7's equals() method, which has satisfactory performance.

It's important to always override the hashCode() method when overriding equals(). I didn't do so in Listing 4-7 because I have yet to formally introduce hashCode().

Finalization

Finalization refers to cleanup via the finalize() method, which is known as a finalizer. The finalize() method's Java documentation states that finalize() is “called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize() method to dispose of system resources or to perform other cleanup.”

Object's version of finalize() does nothing; you must override this method with any needed cleanup code. Because the virtual machine might never call finalize() before an application terminates, you should provide an explicit cleanup method and have finalize() call this method as a safety net in case the method isn't otherwise called.
Caution Never depend on finalize() for releasing limited resources such as file descriptors. For example, if an application object opens files, expecting that its finalize() method will close them, the application might find itself unable to open additional files when a tardy virtual machine is slow to call finalize(). What makes this problem worse is that finalize() might be called more frequently on another virtual machine, resulting in this too-many-open-files problem not revealing itself. The developer might falsely believe that the application behaves consistently across different virtual machines.

If you decide to override finalize(), your object’s subclass layer must give its superclass layer an opportunity to perform finalization. You can accomplish this task by specifying super.finalize(); as the last statement in your method, which the following example demonstrates:

```java
@Override
protected void finalize() throws Throwable
{
    try
    {
        // Perform subclass cleanup.
    }
    finally
    {
        super.finalize();
    }
}
```

The example's finalize() declaration appends throws Throwable to the method header because the cleanup code might throw an exception. If an exception is thrown, execution leaves the method and, in the absence of try-finally, super.finalize(); never executes. (I will discuss exceptions and try-finally in Chapter 5.)

To guard against this possibility, the subclass’s cleanup code executes in a block that follows reserved word try. If an exception is thrown, Java’s exception-handling logic executes the block following the finally reserved word, and super.finalize(); executes the superclass’s finalize() method.

Note The finalize() method has often been used to perform resurrection (making an unreferenced object referenced) to implement object pools that recycle the same objects when these objects are expensive (time-wise) to create (database connection objects are an example).

Resurrection occurs when you assign this (a reference to the current object) to a class or instance field (or to another long-lived variable). For example, you might specify r = this; within finalize() to assign the unreferenced object identified as this to a class field named r.

Because of the possibility for resurrection, there is a severe performance penalty imposed on the garbage collection of an object that overrides finalize().

A resurrected object’s finalizer cannot be called again.
Hash Codes

The `hashCode()` method returns a 32-bit integer that identifies the current object's hash code, a small value that results from applying a mathematical function to a potentially large amount of data. The calculation of this value is known as hashing.

You must override `hashCode()` when overriding `equals()` and in accordance with the following contract, which is specified in `hashCode()`'s Java documentation:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals(Object)` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects might improve the performance of hash tables.

Fail to obey this contract and your class's instances will not work properly with Java's hash-based Collections Framework classes, such as `java.util.HashMap`. (I will discuss `HashMap` and other Collections Framework classes in Chapter 9.)

If you override `equals()` but not `hashCode()`, you most importantly violate the second item in the contract: the hash codes of equal objects must also be equal. This violation can lead to serious consequences, as demonstrated in the following example:

```java
java.util.Map<
    Point,
    String>
map = new java.util.HashMap<
    Point,
    String>();
map.put(p1, "first point");
System.out.println(map.get(p1)); // Output: first point
System.out.println(map.get(new Point(10, 20))); // Output: null
```

Assume that the example's statements are appended to Listing 4-7's `main()` method—the `java.util` prefix and `<Point, String>` have to do with packages and generics, which I discuss in Chapters 5 and 6.

After `main()` creates its `Point` objects and calls its `System.out.println()` methods, it executes the example's statements, which perform the following tasks:

- The first statement instantiates `HashMap`, which is in the `java.util` package.
- The second statement calls `HashMap`'s `put()` method to store Listing 4-7's `p1` object key and the "first point" value in the hashmap.
- The third statement retrieves the value of the hashmap entry whose `Point` key is logically equal to `p1` via `HashMap`'s `get()` method.
- The fourth statement is equivalent to the third statement but returns the null reference instead of "first point."
Although objects p1 and Point(10, 20) are logically equivalent, these objects have different hash codes, resulting in each object referring to a different entry in the hashmap. If an object is not stored (via put()) in that entry, get() returns null.

Correcting this problem requires that hashCode() be overridden to return the same integer value for logically equivalent objects. I will show you how to accomplish this task when I discuss HashMap in Chapter 9.

**String Representation**

The toString() method returns a string-based representation of the current object. This representation defaults to the object’s class name, followed by the @ symbol, followed by a hexadecimal representation of the object’s hash code.

For example, if you were to execute System.out.println(p1); to output Listing 4-7’s p1 object, you would see a line of output similar to Point@3e25a5. (System.out.println() calls p1’s inherited toString() method behind the scenes.)

You should strive to override toString() so that it returns a concise but meaningful description of the object. For example, you might declare, in Listing 4-7’s Point class, a toString() method that is similar to the following:

```java
@Override
public String toString()
{
    return "(" + x + ", " + y + ")";
}
```

This time, executing System.out.println(p1); results in more meaningful output, such as (10, 20).

**Composition**

Implementation inheritance and composition offer two different approaches to reusing code. As you have learned, implementation inheritance is concerned with extending a class with a new class, which is based upon an “is-a” relationship between them: a Car is a Vehicle, for example.

On the other hand, composition is concerned with composing classes out of other classes, which is based upon a “has-a” relationship between them. For example, a Car has an Engine, Wheels, and a SteeringWheel.

You have already seen examples of composition in this chapter. For example, Listing 4-2’s Car class includes String make and String model fields. Listing 4-8’s Car class provides another example of composition.
Listing 4-8. A Car Class Whose Instances Are Composed of Other Objects

class Car extends Vehicle
{
    private Engine engine; // bicycles don't have engines
    private Wheel[] wheels; // boats don't have wheels
    private SteeringWheel steeringWheel; // hang gliders don't have steering wheels
}

Listing 4-8 demonstrates that composition and implementation inheritance are not mutually exclusive. Although not shown, Car inherits various members from its Vehicle superclass, in addition to providing its own engine, wheels, and steeringWheel fields.

The Trouble with Implementation Inheritance

Implementation inheritance is potentially dangerous, especially when the developer doesn’t have complete control over the superclass or when the superclass isn’t designed and documented with extension in mind.

The problem is that implementation inheritance breaks encapsulation. The subclass relies on implementation details in the superclass. If these details change in a new version of the superclass, the subclass might break, even when the subclass isn’t touched.

For example, suppose you have purchased a library of Java classes, and one of these classes describes an appointment calendar. Although you don’t have access to this class’s source code, assume that Listing 4-9 describes part of its code.

Listing 4-9. An Appointment Calendar Class

class ApptCalendar
{
    private final static int MAX_APPT = 1000;
    private Appt[] appts;
    private int size;

    public ApptCalendar()
    {
        appts = new Appt[MAX_APPT];
        size = 0; // redundant because field automatically initialized to 0
        // adds clarity, however
    }

    public void addAppt(Appt appt)
    {
        if (size == appts.length)
            return; // array is full
        appts[size++] = appt;
    }

```java
public void addAppts(Appt[] appts)
{
    for (int i = 0; i < appts.length; i++)
        addAppt(appts[i]);
}
}
```

Listing 4-9’s ApptCalendar class stores an array of appointments, with each appointment described by an Appt instance. For this discussion, the details of Appt are irrelevant. It could be as trivial as class Appt {}.

Suppose you want to log each appointment in a file. Because a logging capability isn’t provided, you extend ApptCalendar with Listing 4-10’s LoggingApptCalendar class, which adds logging behavior in overriding addAppt() and addAppts() methods.

**Listing 4-10. Extending the Appointment Calendar Class**

```java
public class LoggingApptCalendar extends ApptCalendar
{
    // A constructor is not necessary because the Java compiler will add a
    // no-argument constructor that calls the superclass’s no-argument
    // constructor by default.

    @Override
    public void addAppt(Appt appt)
    {
        Logger.log(appt.toString());
        super.addAppt(appt);
    }

    @Override
    public void addAppts(Appt[] appts)
    {
        for (int i = 0; i < appts.length; i++)
            Logger.log(appts[i].toString());
        super.addAppts(appts);
    }
}
```

Listing 4-10’s LoggingApptCalendar class relies on a Logger class whose void log(String msg) class method logs a string to a file (the details are unimportant). Notice the use of toString() to convert an Appt object to a String object, which is then passed to log().

Although this class looks okay, it doesn’t work as you might expect. Suppose you instantiate this class and add a few Appt instances to this instance via addAppts(), as demonstrated in the following manner:

```java
LoggingApptCalendar lapptc = new LoggingApptCalendar();
lapptc.addAppts(new Appt[] { new Appt(), new Appt(), new Appt() });
```
If you also add a `System.out.println(msg)` method call to Logger's `log(String msg)` method, to output this method's argument, you will discover that `log()` outputs a total of six messages; each of the expected three messages (one per `Appt` object) is duplicated.

When `LoggingApptCalendar`'s `addAppts()` method is called, it first calls `Logger.log()` for each `Appt` instance in the `appts` array that is passed to `addAppts()`. This method then calls `ApptCalendar`'s `addAppts()` method via `super.addAppts(appts);`.

`ApptCalendar`'s `addAppts()` method calls `LoggingApptCalendar`'s overriding `addAppt()` method for each `Appt` instance in its `appts` array argument. `addAppt()` executes `Logger.log(appt.toString());` to log its `appt` argument's string representation, and you end up with three additional logged messages.

If you didn't override the `addAppts()` method, this problem would go away. However, the subclass would be tied to an implementation detail: `ApptCalendar`'s `addAppts()` method calls `addAppt()`.

It isn’t a good idea to rely on an implementation detail when the detail isn’t documented. I previously stated that you don’t have access to `ApptCalendar`'s source code.) When a detail isn’t documented, it can change in a new version of the class.

Because a base class change can break a subclass, this problem is known as the **fragile base class problem**. A related cause of fragility that also has to do with overriding methods occurs when new methods are added to a superclass in a subsequent release.

For example, suppose a new version of the library introduces a new `public void addAppt(Appt appt, boolean unique)` method into the `ApptCalendar` class. This method adds the `appt` instance to the calendar when `unique` is false; and, when `unique` is true, it adds the `appt` instance only if it has not previously been added.

Because this method has been added after the `LoggingApptCalendar` class was created, `LoggingApptCalendar` doesn’t override the new `addAppt()` method with a call to `Logger.log()`. As a result, `Appt` instances passed to the new `addAppt()` method are not logged.

Here is another problem: you introduce a method into the subclass that is not also in the superclass. A new version of the superclass presents a new method that matches the subclass method signature and return type. Your subclass method now overrides the superclass method and probably doesn’t fulfill the superclass method’s contract.

There is a way to make these problems disappear. Instead of extending the superclass, create a private field in a new class, and have this field reference an instance of the superclass. This task demonstrates composition because you are forming a “has-a” relationship between the new class and the superclass.

Additionally, have each of the new class’s instance methods call the corresponding superclass method via the superclass instance that was saved in the private field, and also return the called method’s return value. This task is known as **forwarding**, and the new methods are known as **forwarding methods**.

Listing 4-11 presents an improved `LoggingApptCalendar` class that uses composition and forwarding to forever eliminate the fragile base class problem and the additional problem of unanticipated method overriding.
CHAPTER 4: Discovering Inheritance, Polymorphism, and Interfaces

Listing 4-11. A Composed Logging Appointment Calendar Class

```java
public class LoggingApptCalendar {
    private ApptCalendar apptCal;

    public LoggingApptCalendar(ApptCalendar apptCal) {
        this.apptCal = apptCal;
    }

    public void addAppt(Appt appt) {
        Logger.log(appt.toString());
        apptCal.addAppt(appt);
    }

    public void addAppts(Appt[] appts) {
        for (int i = 0; i < appts.length; i++)
            Logger.log(appts[i].toString());
        apptCal.addAppts(appts);
    }
}
```

Listing 4-11’s LoggingApptCalendar class doesn’t depend upon implementation details of the ApptCalendar class. You can add new methods to ApptCalendar and they will not break LoggingApptCalendar.

**Note**   LoggingApptCalendar is an example of a *wrapper class*, a class whose instances wrap other instances. Each LoggingApptCalendar instance wraps an ApptCalendar instance. LoggingApptCalendar is also an example of the *Decorator design pattern*, which is presented on page 175 of *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; ISBN: 0201633612).

When should you extend a class and when should you use a wrapper class? Extend a class when an “is-a” relationship exists between the superclass and the subclass, and either you have control over the superclass or the superclass has been designed and documented for class extension. Otherwise, use a wrapper class.

What does “design and document for class extension” mean? “Design” means provide protected methods that hook into the class’s inner workings (to support writing efficient subclasses) and ensure that constructors and the `clone()` method never call overridable methods. “Document” means clearly state the impact of overriding methods.
Wrapper classes shouldn’t be used in a callback framework, an object framework in which an object passes its own reference to another object (via this) so that the latter object can call the former object’s methods at a later time. This “calling back to the former object’s method” is known as a callback. Because the wrapped object doesn’t know of its wrapper class, it passes only its reference (via this), and resulting callbacks don’t involve the wrapper class’s methods.

Caution

Changing Form

Some real-world entities can change their forms. For example, water (on Earth as opposed to interstellar space) is normally a liquid, but it changes to a solid when frozen, and it changes to a gas when heated to its boiling point. Insects such as butterflies that undergo metamorphosis are another example.

The ability to change form is known as polymorphism and is useful to model in a programming language. For example, code that draws arbitrary shapes can be expressed more concisely by introducing a single Shape class and its draw() method and by invoking that method for each Circle instance, Rectangle instance, and other Shape instance stored in an array. When Shape’s draw() method is called for an array instance, it is the Circle’s, Rectangle’s or other Shape instance’s draw() method that gets called. There are many forms of Shape’s draw() method. In other words, this method is polymorphic.

Java supports four kinds of polymorphism:

- **Coercion**: An operation serves multiple types through implicit type conversion. For example, division lets you divide an integer by another integer or divide a floating-point value by another floating-point value. If one operand is an integer and the other operand is a floating-point value, the compiler coerces (implicitly converts) the integer to a floating-point value to prevent a type error. (There is no division operation that supports an integer operand and a floating-point operand.) Passing a subclass object reference to a method’s superclass parameter is another example of coercion polymorphism. The compiler coerces the subclass type to the superclass type to restrict operations to those of the superclass.

- **Overloading**: The same operator symbol or method name can be used in different contexts. For example, + can be used to perform integer addition, floating-point addition, or string concatenation, depending on the types of its operands. Also, multiple methods having the same name can appear in a class (through declaration and/or inheritance).

- **Parametric**: Within a class declaration, a field name can associate with different types and a method name can associate with different parameter and return types. The field and method can then take on different types in each class instance. For example, a field might be of type java.lang.Integer and a method might return an Integer in one class instance, and the same field might be of type String and the same method might return a String in another class instance. Java supports parametric polymorphism via generics, which I will discuss in Chapter 6.
Subtype: A type can serve as another type’s subtype. When a subtype instance appears in a supertype context, executing a supertype operation on the subtype instance results in the subtype’s version of that operation executing. For example, suppose that Circle is a subclass of Point and that both classes contain a draw() method. Assigning a Circle instance to a variable of type Point, and then calling the draw() method via this variable, results in Circle’s draw() method being called.

Many developers don’t regard coercion and overloading as valid kinds of polymorphism. They see coercion and overloading as nothing more than type conversions and syntactic sugar. In contrast, parametric and subtype are regarded as valid kinds of polymorphism.

In this section, I focus on subtype polymorphism by first examining upcasting and late binding. I then introduce you to abstract classes and abstract methods, downcasting and runtime type identification, and covariant return types.

### Upcasting and Late Binding

Listing 4-7’s Point class represents a point as an x-y pair. Because a circle (in this example) is an x-y pair denoting its center, and has a radius denoting its extent, you can extend Point with a Circle class that introduces a radius field. Check out Listing 4-12.

**Listing 4-12. A Circle Class Extending the Point Class**

class Circle extends Point
{
    private int radius;

    Circle(int x, int y, int radius)
    {
        super(x, y);
        this.radius = radius;
    }

    int getRadius()
    {
        return radius;
    }

    @Override
    public String toString()
    {
        return "" + radius;
    }
}
Listing 4-12’s Circle class describes a Circle as a Point with a radius, which implies that you can treat a Circle instance as if it was a Point instance. Accomplish this task by assigning the Circle instance to a Point variable, as demonstrated here:

```java
Circle c = new Circle(10, 20, 30);
Point p = c;
```

The cast operator isn’t needed to convert from Circle to Point because access to a Circle instance via Point’s interface is legal. After all, a Circle is at least a Point. This assignment is known as upcasting because you are implicitly casting up the type hierarchy (from the Circle subclass to the Point superclass).

After upcasting Circle to Point, you cannot call Circle’s getRadius() method because this method is not part of Point’s interface. Losing access to subtype features after narrowing a subclass to its superclass seems useless but is necessary for achieving subtype polymorphism.

In addition to upcasting the subclass instance to a variable of the superclass type, subtype polymorphism involves declaring a method in the superclass and overriding this method in the subclass. For example, suppose Point and Circle are to be part of a graphics application, and you need to introduce a `draw()` method into each class to draw a point and a circle, respectively. You end with the class structure shown in Listing 4-13.

**Listing 4-13. Declaring a Graphics Application’s Point and Circle Classes**

```java
class Point
{
    private int x, y;

    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    int getX()
    {
        return x;
    }

    int getY()
    {
        return y;
    }

    @Override
    public String toString()
    {
        return "(" + x + ", " + y + ")";
    }

    void draw()
    {
        System.out.println("Point drawn at " + toString());
    }
}
```
class Circle extends Point {
    private int radius;

    Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    int getRadius() {
        return radius;
    }

    @Override
    public String toString() {
        return radius;
    }

    @Override
    void draw() {
        System.out.println("Circle drawn at " + super.toString() + " with radius " + toString());
    }
}

Listing 4-13’s draw() methods will ultimately draw graphics shapes, but simulating their behaviors via System.out.println() method calls is sufficient during the early testing phase of the graphics application.

Now that you have temporarily finished with Point and Circle, you will want to test their draw() methods in a simulated version of the graphics application. To achieve this objective, you write Listing 4-14’s Graphics class.

Listing 4-14. A Graphics Class for Testing Point’s and Circle’s draw() Methods

public class Graphics {
    public static void main(String[] args) {
        Point[] points = new Point[] { new Point(10, 20), new Circle(10, 20, 30));
        for (int i = 0; i < points.length; i++)
            points[i].draw();
    }
}
Listing 4-14’s `main()` method first declares an array of Points. Upcasting is demonstrated by first having the array’s initializer instantiate the `Circle` class and then by assigning this instance’s reference to the second element in the `points` array.

Moving on, `main()` uses a for loop to call each Point element’s `draw()` method. Because the first iteration calls Point’s `draw()` method, whereas the second iteration calls Circle’s `draw()` method, you observe the following output:

```
Point drawn at (10, 20)
Circle drawn at (10, 20) with radius 30
```

How does Java “know” that it must call Circle’s `draw()` method on the second loop iteration? Should it not call Point’s `draw()` method because Circle is being treated as a Point thanks to the upcast?

At compile time, the compiler doesn’t know which method to call. All it can do is verify that a method exists in the superclass, and verify that the method call’s arguments list and return type match the superclass’s method declaration.

In lieu of knowing which method to call, the compiler inserts an instruction into the compiled code that, at runtime, fetches and uses whatever reference is in `points[i]` to call the correct `draw()` method. This task is known as *late binding*.

Late binding is used for calls to non-final instance methods. For all other method calls, the compiler knows which method to call and inserts an instruction into the compiled code that calls the method associated with the variable’s type (not its value). This task is known as *early binding*.

You can also upcast from one array to another provided that the array being upcast is a subtype of the other array. Consider Listing 4-15.

```
Listing 4-15. Demonstrating Array Upcasting

class Point
{
    private int x, y;

    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    int getX() { return x; }
    int getY() { return y; }
}

class ColoredPoint extends Point
{
    private int color;

    ColoredPoint(int x, int y, int color)
    {
```
super(x, y);
    this.color = color;
}

int getColor() { return color; }
}

public class UpcastArrayDemo
{
    public static void main(String[] args)
    {
        ColoredPoint[] cptArray = new ColoredPoint[1];
        cptArray[0] = new ColoredPoint(10, 20, 5);
        Point[] ptArray = cptArray;
        System.out.println(ptArray[0].getX()); // Output: 10
        System.out.println(ptArray[0].getY()); // Output: 20
        // System.out.println(ptArray[0].getColor()); // Illegal
    }
}

Listing 4-15's main() method first creates a ColoredPoint array consisting of one element. It then instantiates this class and assigns the object’s reference to this element. Because ColoredPoint[] is a subtype of Point[], main() is able to upcast cptArray’s ColoredPoint[] type to Point[] and assign its reference to ptArray.

main() then invokes the ColoredPoint instance’s getX() and getY() methods via ptArray[0]. It cannot invoke getColor() because ptArray has narrower scope than cptArray. In other words, getColor() is not part of Point’s interface.

Abstract Classes and Abstract Methods

Suppose new requirements dictate that your graphics application must include a Rectangle class. Furthermore, this class must include a draw() method, and this method must be tested in a manner similar to that shown in Listing 4-14’s Graphics application class.

In contrast to Circle, which is a Point with a radius, it doesn’t make sense to think of a Rectangle as being a Point with a width and height. Rather, a Rectangle instance would probably be composed of a Point instance indicating its origin and a Point instance indicating its width and height extents.

Because circles, points, and rectangles are examples of shapes, it makes more sense to declare a Shape class with its own draw() method than to specify class Rectangle extends Point. Listing 4-16 presents Shape’s declaration.

Listing 4-16. Declaring a Shape Class

class Shape
{
    void draw()
    {
    }
}
Listing 4-16’s Shape class declares an empty draw() method that only exists to be overridden and to demonstrate subtype polymorphism.

You can now refactor Listing 4-13’s Point class to extend Listing 4-16’s Shape class, leave Circle as is, and introduce a Rectangle class that extends Shape. You can then refactor Listing 4-14’s Graphics class’s main() method to take Shape into account. Listing 4-17 presents the resulting Graphics class.

Listing 4-17. A Graphics Class with a New main() Method That Takes Shape into Account

```java
public class Graphics
{
    public static void main(String[] args)
    {
        Shape[] shapes = new Shape[] { new Point(10, 20), new Circle(10, 20, 30),
                                       new Rectangle(20, 30, 15, 25) };
        for (int i = 0; i < shapes.length; i++)
            shapes[i].draw();
    }
}
```

Because Point and Rectangle directly extend Shape, and because Circle indirectly extends Shape by extending Point, Listing 4-17’s main() method will call the appropriate subclass’s draw() method in response to shapes[i].draw();.

Although Shape makes the code more flexible, there is a problem. What is to stop the developer from instantiating Shape and adding this meaningless instance to the shapes array, as follows?

```java
Shape[] shapes = new Shape[] { new Point(10, 20), new Circle(10, 20, 30),
                               new Rectangle(20, 30, 15, 25), new Shape() };
```

What does it mean to instantiate Shape? Because this class describes an abstract concept, what does it mean to draw a generic shape? Fortunately, Java provides a solution to this problem, which is demonstrated in Listing 4-18.

Listing 4-18. Abstracting the Shape Class

```java
abstract class Shape
{
    abstract void draw(); // semicolon is required
}
```

Listing 4-18 uses Java’s abstract reserved word to declare a class that cannot be instantiated. The compiler reports an error when you try to instantiate this class.

Tip Get into the habit of declaring classes that describe generic categories (such as shape, animal, vehicle, and account) abstract. This way, you will not inadvertently instantiate them.
The abstract reserved word is also used to declare a method without a body. The `draw()` method doesn’t need a body because it cannot draw an abstract shape.

**Caution** The compiler reports an error when you attempt to declare a class that is both abstract and final. For example, `abstract final class Shape` is an error because an abstract class cannot be instantiated and a final class cannot be extended. The compiler also reports an error when you declare a method to be abstract but do not declare its class to be abstract. For example, removing `abstract` from the `Shape` class’s header in Listing 4-18 results in an error. This removal is an error because a non-abstract (concrete) class cannot be instantiated when it contains an abstract method. Finally, when you extend an abstract class, the extending class must override all of the abstract class’s abstract methods, or else the extending class must itself be declared to be abstract; otherwise, the compiler will report an error.

An abstract class can contain non-abstract methods in addition to or instead of abstract methods. For example, Listing 4-2’s `Vehicle` class could have been declared abstract. The constructor would still be present, to initialize private fields, even though you could not instantiate the resulting class.

**Downcasting and Runtime Type Identification**

Moving up the type hierarchy, via upcasting, causes loss of access to subtype features. For example, assigning a `Circle` instance to a `Point` variable `p` means that you cannot use `p` to call `Circle`’s `getRadius()` method.

However, it is possible to once again access the `Circle` instance’s `getRadius()` method by performing an explicit cast operation, for example, `Circle c = (Circle) p;`. This assignment is known as *downcasting* because you are explicitly moving down the type hierarchy (from the `Point` superclass to the `Circle` subclass).

Although an upcast is always safe (the superclass’s interface is a subset of the subclass’s interface), the same cannot be said of a downcast. Listing 4-19 shows you what kind of trouble you can get into when downcasting is used incorrectly.

**Listing 4-19. The Trouble with Downcasting**

```java
public class DowncastDemo {

    class A {
    
    }

    class B extends A {
        void m() {
            
        }
    }

    public class DowncastDemo {
    
```
public static void main(String[] args) {
    A a = new A();
    B b = (B) a;
    b.m();
}

Listing 4-19 presents a class hierarchy consisting of a superclass named A and a subclass named B. Although A doesn’t declare any members, B declares a single m() method.

A third class named DowncastDemo provides a main() method that first instantiates A, and then tries to downcast this instance to B and assign the result to variable b. The compiler will not complain because downcasting from a superclass to a subclass in the same type hierarchy is legal.

However, if the assignment is allowed, the application will undoubtedly crash when it tries to execute b.m();. The crash happens because the virtual machine will attempt to call a method that doesn’t exist—class A doesn’t have an m() method.

Fortunately, this scenario will never happen because the virtual machine verifies that the cast is legal before performing the cast operation. Because it detects that A doesn’t have an m() method, it doesn’t permit the cast by throwing an instance of the ClassCastException class.

The virtual machine’s cast verification illustrates runtime type identification (or RTTI, for short). Cast verification performs RTTI by examining the type of the cast operator’s operand to see whether the cast should be allowed or not. Clearly, the cast should not be allowed.

A second form of RTTI involves the instanceof operator. This operator checks the left operand to see whether or not it is an instance of the right operand and returns true if this is the case. The following example introduces instanceof to Listing 4-19 to prevent the ClassCastException:

if(a instanceof B) {
    B b = (B) a;
    b.m();
}

The instanceof operator detects that variable a’s instance was not created from B and returns false to indicate this fact. As a result, the code that performs the illegal cast will not execute. (Overuse of instanceof probably indicates poor software design.)

Because a subtype is a kind of supertype, instanceof will return true when its left operand is a subtype instance or a supertype instance of its right operand supertype. The following example demonstrates this:

A a = new A();
B b = new B();
System.out.println(b instanceof A); // Output: true
System.out.println(a instanceof A); // Output: true
This example assumes the class structure shown in Listing 4-19 and instantiates superclass A and subclass B. The first `System.out.println()` method call outputs `true` because b’s reference identifies an instance of a subclass of A; the second `System.out.println()` method call outputs `true` because a’s reference identifies an instance of superclass A.

You can also downcast from one array to another provided that the array being downcast is a supertype of the other array, and its elements types are those of the subtype. Consider Listing 4-20.

**Listing 4-20. Demonstrating Array Downcasting**

class Point {
    private int x, y;

    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    int getX() { return x; }
    int getY() { return y; }
}

class ColoredPoint extends Point {
    private int color;

    ColoredPoint(int x, int y, int color)
    {
        super(x, y);
        this.color = color;
    }

    int getColor() { return color; }
}

public class DowncastArrayDemo {
    public static void main(String[] args)
    {
        ColoredPoint[] cptArray = new ColoredPoint[1];
        cptArray[0] = new ColoredPoint(10, 20, 5);
        Point[] ptArray = cptArray;
        System.out.println(ptArray[0].getX()); // Output: 10
        System.out.println(ptArray[0].getY()); // Output: 20
        // System.out.println(ptArray[0].getColor()); // Illegal
        if (ptArray instanceof ColoredPoint[])
        {
            ColoredPoint cp = (ColoredPoint) ptArray[0];
            System.out.println(cp.getColor());
        }
    }
}
Listing 4-20 is similar to Listing 4-15 except that it also demonstrates downcasting. Notice its use of instanceof to verify that ptArray's referenced object is of type ColoredPoint[]. If this operator returns true, it is safe to downcast ptArray[0] from Point to ColoredPoint and assign the reference to ColoredPoint.

**Covariant Return Types**

A covariant return type is a method return type that, in the superclass’s method declaration, is the supertype of the return type in the subclass’s overriding method declaration. Listing 4-21 provides a demonstration of this language feature.

*Listing 4-21. A Demonstration of Covariant Return Types*

class SuperReturnType{
   @Override
   public String toString()
   {
      return "superclass return type";
   }
}

class SubReturnType extends SuperReturnType{
   @Override
   public String toString()
   {
      return "subclass return type";
   }
}

class Superclass{
   SuperReturnType createReturnType()
   {
      return new SuperReturnType();
   }
}

class Subclass extends Superclass{
   @Override
   SubReturnType createReturnType()
   {
      return new SubReturnType();
   }
}

public class CovarDemo{

public static void main(String[] args)
{
    SuperReturnType suprt = new Superclass().createReturnType();
    System.out.println(suprt); // Output: superclass return type
    SubReturnType subrt = new Subclass().createReturnType();
    System.out.println(subrt); // Output: subclass return type
}

Listing 4-21 declares SuperReturnType and Superclass superclasses and SubReturnType and Subclass subclasses; each of Superclass and Subclass declares a createReturnType() method. Superclass's method has its return type set to SuperReturnType, whereas Subclass's overriding method has its return type set to SubReturnType, a subclass of SuperReturnType.

Covariant return types minimize upcasting and downcasting. For example, Subclass's createReturnType() method doesn't need to upcast its SubReturnType instance to its SubReturnType return type. Furthermore, this instance doesn't need to be downcast to SubReturnType when assigning to variable subrt.

In the absence of covariant return types, you would end up with Listing 4-22.

Listing 4-22. Upcasting and Downcasting in the Absence of Covariant Return Types

class SuperReturnType
{
    @Override
    public String toString()
    {
        return "superclass return type";
    }
}

class SubReturnType extends SuperReturnType
{
    @Override
    public String toString()
    {
        return "subclass return type";
    }
}

class Superclass
{
    SuperReturnType createReturnType()
    {
        return new SuperReturnType();
    }
}
class Subclass extends Superclass
{
    @Override
    SuperReturnType createReturnType()
    {
        return new SubReturnType();
    }
}

public class CovarDemo
{
    public static void main(String[] args)
    {
        SuperReturnType suprt = new Superclass().createReturnType();
        System.out.println(suprt); // Output: superclass return type
        SubReturnType subrt = (SubReturnType) new Subclass().createReturnType();
        System.out.println(subrt); // Output: subclass return type
    }
}

In Listing 4-22, the first bolded code reveals an upcast from SubReturnType to SuperReturnType, and the second bolded code uses the required (SubReturnType) cast operator to downcast from SuperReturnType to SubReturnType, prior to the assignment to subrt.

**Formalizing Class Interfaces**

In my introduction to information hiding (see Chapter 3), I stated that every class $X$ exposes an interface (a protocol consisting of constructors, methods, and [possibly] fields that are made available to objects created from other classes for use in creating and communicating with $X$'s objects).

Java formalizes the interface concept by providing reserved word interface, which is used to introduce a type without implementation. Java also provides language features to declare, implement, and extend interfaces. After looking at interface declaration, implementation, and extension in this section, I explain the rationale for using interfaces.

**Declaring Interfaces**

An interface declaration consists of a header followed by a body. At minimum, the header consists of reserved word interface followed by a name that identifies the interface. The body starts with an open brace character and ends with a close brace. Sandwiched between these delimiters are constant and method header declarations. Consider Listing 4-23.
Listing 4-23. Declaring a Drawable Interface

```java
interface Drawable {
    int RED = 1;   // For simplicity, integer constants are used. These constants are
    int GREEN = 2; // not that descriptive, as you will see.
    int BLUE = 3;
    int BLACK = 4;
    void draw(int color);
}
```

Listing 4-23 declares an interface named Drawable. By convention, an interface’s name begins with an uppercase letter. Furthermore, the first letter of each subsequent word in a multiword interface name is capitalized.

Note Many interface names end with the able suffix. For example, the standard class library includes interfaces named Callable, Comparable, Cloneable, Iterable, Runnable, and Serializable. It is not mandatory to use this suffix; the standard class library also provides interfaces named CharSequence, Collection, Executor, Future, Iterator, List, Map, and Set.

Drawable declares four fields that identify color constants. Drawable also declares a draw() method that must be called with one of these constants to specify the color used to draw something.

Note You can precede interface with public to make your interface accessible to code outside of its package. (I will discuss packages in the next chapter). Otherwise, the interface is only accessible to other types in its package. You can also precede interface with abstract, to emphasize that an interface is abstract. Because an interface is already abstract, it is redundant to specify abstract in the interface’s declaration. An interface’s fields are implicitly declared public, static, and final. It is therefore redundant to declare them with these reserved words. Because these fields are constants, they must be explicitly initialized; otherwise, the compiler reports an error. Finally, an interface’s methods are implicitly declared public and abstract. Therefore, it is redundant to declare them with these reserved words. Because these methods must be instance methods, don’t declare them static or the compiler will report errors.

Drawable identifies a type that specifies what to do (draw something) but not how to do it. It leaves implementation details to classes that implement this interface. Instances of such classes are known as drawables because they know how to draw themselves.
An interface that declares no members is known as a **marker interface** or a **tagging interface**. It associates metadata with a class. For example, the presence of the Cloneable marker/tagging interface implies that instances of its implementing class can be shallowly cloned. RTTI is used to detect that an object's class implements a marker/tagging interface. For example, when Object's `clone()` method detects, via RTTI, that the calling instance's class implements Cloneable, it shallowly clones the object.

### Implementing Interfaces

By itself, an interface is useless. To be of any benefit to an application, the interface needs to be implemented by a class. Java provides the `implements` reserved word for this task. This reserved word is demonstrated in Listing 4-24.

#### Listing 4-24. Implementing the **Drawable** Interface

class Point implements Drawable {
    private int x, y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    @Override
    public void draw(int color) {
        System.out.println("Point drawn at " + toString() + " in color " + color);
    }
}
class Circle extends Point implements Drawable
{
    private int radius;

    Circle(int x, int y, int radius)
    {
        super(x, y);
        this.radius = radius;
    }

    int getRadius()
    {
        return radius;
    }

    @Override
    public String toString()
    {
        return "" + radius;
    }

    @Override
    public void draw(int color)
    {
        System.out.println("Circle drawn at " + super.toString() + " with radius " + toString() + " in color " + color);
    }
}

Listing 4-24 retrofits Listing 4-13’s class hierarchy to take advantage of Listing 4-23’s Drawable interface. You will notice that each of classes Point and Circle implements this interface by attaching the implements Drawable clause to its class header.

To implement an interface, the class must specify, for each interface method header, a method whose header has the same signature and return type as the interface’s method header and a code body to go with the method header.

Caution When implementing a method, don’t forget that the interface’s methods are implicitly declared public. If you forget to include public in the implemented method’s declaration, the compiler will report an error because you are attempting to assign weaker access to the implemented method.

When a class implements an interface, the class inherits the interface’s constants and method headers and overrides the method headers by providing implementations (hence the @Override annotation). This is known as interface inheritance.
It turns out that Circle’s header doesn’t need the implements Drawable clause. If this clause is not present, Circle inherits Point’s draw() method and is still considered to be a Drawable, whether it overrides this method or not.

An interface specifies a type whose data values are the objects whose classes implement the interface and whose behaviors are those specified by the interface. This fact implies that you can assign an object’s reference to a variable of the interface type, provided that the object’s class implements the interface. The following example provides a demonstration:

```java
public static void main(String[] args) {
    Drawable[] drawables = new Drawable[] { new Point(10, 20), new Circle(10, 20, 30) };
    for (int i = 0; i < drawables.length; i++)
        drawables[i].draw(Drawable.RED);
}
```

Because Point and Circle instances are drawables by virtue of these classes implementing the Drawable interface, it is legal to assign Point and Circle instance references to variables (including array elements) of type Drawable.

When you run this method, it generates the following output:

Point drawn at (10, 20) in color 1
Circle drawn at (10, 20) with radius 30 in color 1

Listing 4-23’s Drawable interface is useful for drawing a shape’s outline. Suppose you also need to fill a shape’s interior. You might attempt to satisfy this requirement by declaring Listing 4-25’s Fillable interface.

### Listing 4-25. Declaring a Fillable Interface

```java
interface Fillable
{
    int RED = 1;
    int GREEN = 2;
    int BLUE = 3;
    int BLACK = 4;
    void fill(int color);
}
```

Given Listings 4-23 and 4-25, you can declare that the Point and Circle classes implement both interfaces by specifying class Point implements Drawable, Fillable and class Circle implements Drawable, Fillable. You can then modify the main() method to also treat the drawables as fillables so that you can fill these shapes, as follows:

```java
public static void main(String[] args) {
    Drawable[] drawables = new Drawable[] { new Point(10, 20),
                                            new Circle(10, 20, 30) };
    for (int i = 0; i < drawables.length; i++)
        drawables[i].draw(Drawable.RED);
```
Fillable[] fillables = new Fillable[drawables.length];
for (int i = 0; i < drawables.length; i++)
{
    fillables[i] = (Fillable) drawables[i];
    fillables[i].fill(Fillable.GREEN);
}

After invoking each drawable's draw() method, main() creates a Fillable array of the same length as the Drawable array. It then proceeds to copy each Drawable array element to a Fillable array element and then invoke the fillable's fill() method. The (Fillable) cast is necessary because a drawable is not a fillable. This cast operation will succeed because the Point and Circle instances being copied implement Fillable as well as Drawable.

**Tip** You can list as many interfaces as you need to implement by specifying a comma-separated list of interface names after implements.

Implementing multiple interfaces can lead to name collisions, and the compiler will report errors. For example, suppose that you attempt to compile Listing 4-26’s interface and class declarations.

Listing 4-26. Colliding Interfaces

```java
interface A
{
    int X = 1;
    void foo();
}

interface B
{
    int X = 1;
    int foo();
}

class Collision implements A, B
{
    @Override
    public void foo();

    @Override
    public int foo() { return X; }
}
```

Each of Listing 4-26’s A and B interfaces declares a constant named X. Despite each constant having the same type and value, the compiler will report an error when it encounters X in Collision’s second foo() method because it doesn't know which X is being inherited.
Speaking of `foo()`, the compiler reports an error when it encounters `Collision`'s second `foo()` declaration because `foo()` has already been declared. You cannot overload a method by changing only its return type.

The compiler will probably report additional errors. For example, the Java 7 compiler has this to say when told to compile Listing 4-26:

Collision.java:19: error: method foo() is already defined in class Collision
    public int foo() { return X; }
^  
Collision.java:13: error: Collision is not abstract and does not override abstract method foo() in B
class Collision implements A, B
^  
Collision.java:16: error: foo() in Collision cannot implement foo() in B
    public void foo();
^  
    return type void is not compatible with int
Collision.java:19: error: reference to X is ambiguous, both variable X in A and variable X in B match
    public int foo() { return X; }
^  
4 errors

Extending Interfaces

Just as a subclass can extend a superclass via reserved word `extends`, you can use this reserved word to have a subinterface extend a superinterface. This, too, is known as interface inheritance.

For example, the duplicate color constants in `Drawable` and `Fillable` lead to name collisions when you specify their names by themselves in an implementing class. To avoid these name collisions, prefix a name with its interface name and the member access operator, or place these constants in their own interface, and have `Drawable` and `Fillable` extend this interface, as demonstrated in Listing 4-27.

Listing 4-27. Extending the Colors Interface

```java
interface Colors {
    int RED = 1;
    int GREEN = 2;
    int BLUE = 3;
    int BLACK = 4;
}

interface Drawable extends Colors {
    void draw(int color);
}

interface Fillable extends Colors {
    ...
}
```
{   void fill(int color);
}

The fact that Drawable and Fillable both inherit constants from Colors is not a problem for the compiler. There is only a single copy of these constants (in Colors) and no possibility of a name collision, and so the compiler is satisfied.

If a class can implement multiple interfaces by declaring a comma-separated list of interface names after implements, it seems that an interface should be able to extend multiple interfaces in a similar way. This feature is demonstrated in Listing 4-28.

Listing 4-28. Extending a Pair of Interfaces

```java
interface A
{
    int X = 1;
}

interface B
{
    double X = 2.0;
}

interface C extends A, B
{
}
```

Listing 4-28 will compile even though C inherits two same-named constants X with different types and initializers. However, if you implement C and then try to access X, as in Listing 4-29, you will run into a name collision.

Listing 4-29. Discovering a Name Collision

```java
class Collision implements C
{
    public void output()
    {
        System.out.println(X); // Which X is accessed?
    }
}
```

Suppose you introduce a void foo(); method header declaration into interface A and an int foo(); method header declaration into interface B. This time, the compiler will report an error when you attempt to compile the modified Listing 4-28.

**Why Use Interfaces?**

Now that the mechanics of declaring, implementing, and extending interfaces are out of the way, you can focus on the rationale for using them. Unfortunately, newcomers to Java’s interfaces feature are often told that this feature was created as a workaround to Java’s lack of support for multiple
implementation inheritance. While interfaces are useful in this capacity, this is not their reason for existence. Instead, Java’s interfaces feature was created to give developers the utmost flexibility in designing their applications by decoupling interface from implementation. You should always code to the interface (supplied by an interface type or an abstract class).

Those who are adherents to agile software development (a group of software development methodologies based on iterative development that emphasizes keeping code simple, testing frequently, and delivering functional pieces of the application as soon as they are deliverable), know the importance of flexible coding. They cannot afford to tie their code to a specific implementation because a change in requirements for the next iteration could result in a new implementation, and they might find themselves rewriting significant amounts of code, which wastes time and slows development.

Interfaces help you achieve flexibility by decoupling interface from implementation. For example, the main() method in Listing 4-17’s Graphics class creates an array of objects from classes that subclass the Shape class, and then iterates over these objects, calling each object’s draw() method. The only objects that can be drawn are those that subclass Shape.

Suppose you also have a hierarchy of classes that model resistors, transistors, and other electronic components. Each component has its own symbol that allows the component to be shown in a schematic diagram of an electronic circuit. Perhaps you want to add a drawing capability to each class that draws that component's symbol.

You might consider specifying Shape as the superclass of the electronic component class hierarchy. However, electronic components are not shapes (although they have shapes) so it makes no sense to place these classes in a class hierarchy rooted in Shape.

However, you can make each component class implement the Drawable interface, which lets you add expressions that instantiate these classes to the drawables array in the main() method appearing prior to Listing 4-25 (so you can draw their symbols). This is legal because these instances are drawables.

Wherever possible, you should strive to specify interfaces instead of classes in your code to keep your code adaptable to change. This is especially true when working with Java’s Collections Framework, which I will discuss at length in Chapter 9.

**Tip** Always strive to specify interfaces instead of classes to keep your code adaptable to change.

For now, consider a simple example that consists of the Collections Framework’s java.util.List interface and its java.util.ArrayList and java.util.LinkedList implementation classes. The following example presents inflexible code based on the ArrayList class:

```java
ArrayList<String> arraylist = new ArrayList<String>();
void dump(ArrayList<String> arraylist)
{
    // suitable code to dump out the arraylist
}
```
This example uses the generics-based parameterized type language feature (which I will discuss in Chapter 6) to identify the kind of objects stored in an ArrayList instance. In this example, String objects are stored.

The example is inflexible because it hardwires the ArrayList class into multiple locations. This hardwiring focuses the developer into thinking specifically about array lists instead of generically about lists.

Lack of focus is problematic when a requirements change, or perhaps a performance issue brought about by profiling (analyzing a running application to check its performance), suggests that the developer should have used LinkedList.

The example only requires a minimal number of changes to satisfy the new requirement. In contrast, a larger code base might need many more changes. Although you only need to change ArrayList to LinkedList, to satisfy the compiler, consider changing ArrayList to LinkedList to keep semantics (meaning) clear—you might have to change multiple occurrences of names that refer to an ArrayList instance throughout the source code.

The developer is bound to lose time while refactoring the code to adapt to LinkedList. Instead, time could have been saved by writing this example to use the equivalent of constants. In other words, the example could have been written to rely on interfaces and to only specify ArrayList in one place. The following example shows you what the resulting code would look like:

```java
List<String> list = new ArrayList<String>();
void dump(List<String> list)
{
    // suitable code to dump out the list
}
```

This example is much more flexible than the previous example. If a requirements or profiling change suggests that LinkedList should be used instead of ArrayList, simply replace Array with Linked and you are done. You don’t even have to change the parameter name.

**Note** Java provides interfaces and abstract classes for describing abstract types (types that cannot be instantiated). Abstract types represent abstract concepts (drawable and shape, for example), and instances of such types would be meaningless.

Interfaces promote flexibility through lack of implementation—Drawable and List illustrate this flexibility. They are not tied to any single class hierarchy but can be implemented by any class in any hierarchy. In contrast, abstract classes support implementation but can be genuinely abstract (Listing 4-18’s abstract Shape class, for example). However, they are limited to appearing in the upper levels of class hierarchies.

Interfaces and abstract classes can be used together. For example, the Collections Framework’s java.util package provides List, Map, and Set interfaces and AbstractList, AbstractMap, and AbstractSet abstract classes that provide skeletal implementations of these interfaces.

By implementing many interface methods, the skeletal implementations make it easy for you to create your own interface implementations, to address your unique requirements. If they don’t meet your needs, you can optionally have your class directly implement the appropriate interface.
The following exercises are designed to test your understanding of Chapter 4’s content:

1. What is implementation inheritance?
2. How does Java support implementation inheritance?
3. Can a subclass have two or more superclasses?
4. How do you prevent a class from being subclassed?
5. True or false: The `super()` call can appear in any method.
6. If a superclass declares a constructor with one or more parameters, and if a subclass constructor doesn’t use `super()` to call that constructor, why does the compiler report an error?
7. What is an immutable class?
8. True or false: A class can inherit constructors.
9. What does it mean to override a method?
10. What is required to call a superclass method from its overriding subclass method?
11. How do you prevent a method from being overridden?
12. Why can you not make an overriding subclass method less accessible than the superclass method it is overriding?
13. How do you tell the compiler that a method overrides another method?
14. Why does Java not support multiple implementation inheritance?
15. What is the name of Java’s ultimate superclass?
16. What is the purpose of the `clone()` method?
17. When does `Object`'s `clone()` method throw `CloneNotSupportedException`?
18. Explain the difference between shallow copying and deep copying.
19. Can the `==` operator be used to determine if two objects are logically equivalent? Why or why not?
20. What does `Object`'s `equals()` method accomplish?
21. Does expression "abc" == "a" + "bc" return true or false?
22. How can you optimize a time-consuming `equals()` method?
23. What is the purpose of the `finalize()` method?
24. Should you rely on `finalize()` for closing open files? Why or why not?
25. What is a hash code?
26. True or false: You should override the `hashCode()` method whenever your override the `equals()` method.
27. What does `Object`'s `toString()` method return?
28. Why should you override toString()?

29. Define composition.

30. True or false: Composition is used to describe “is-a” relationships and implementation inheritance is used to describe “has-a” relationships.

31. Identify the fundamental problem of implementation inheritance. How do you fix this problem?

32. Define subtype polymorphism.

33. How is subtype polymorphism accomplished?

34. Why would you use abstract classes and abstract methods?

35. Can an abstract class contain concrete methods?

36. What is the purpose of downcasting?

37. List two forms of RTTI.

38. What is a covariant return type?

39. How do you formally declare an interface?

40. True or false: You can precede an interface declaration with the abstract reserved word.

41. Define marker interface.

42. What is interface inheritance?

43. How do you implement an interface?

44. What problem might you encounter when you implement multiple interfaces?

45. How do you form a hierarchy of interfaces?

46. Why is Java’s interfaces feature so important?

47. What do interfaces and abstract classes accomplish?

48. How do interfaces and abstract classes differ?

49. Model part of an animal hierarchy by declaring Animal, Bird, Fish, AmericanRobin, DomesticCanary, RainbowTrout, and SockeyeSalmon classes:

   - Animal is public and abstract, declares private String-based kind and appearance fields, declares a public constructor that initializes these fields to passed-in arguments, declares public and abstract eat() and move() methods that take no arguments and whose return type is void, and overrides the toString() method to output the contents of kind and appearance.

   - Bird is public and abstract, extends Animal, declares a public constructor that passes its kind and appearance parameter values to its superclass constructor, overrides its eat() method to output eats seeds and insects (via System.out.println()), and overrides its move() method to output flies through the air.

   - Fish is public and abstract; extends Animal; declares a public constructor that passes its kind and appearance parameter values to its superclass constructor; overrides its eat() method to output eats krill, algae, and insects; and overrides its move() method to output swims through the water.
• AmericanRobin is public, extends Bird, and declares a public noargument constructor that passes "americanrobin" and "red breast" to its superclass constructor.

• DomesticCanary is public, extends Bird, and declares a public noargument constructor that passes "domesticcanary" and "yellow, orange, black, brown, white, red" to its superclass constructor.

• RainbowTrout is public, extends Fish, and declares a public noargument constructor that passes "rainbowtrout" and "bands of brilliant speckled multicolored stripes running nearly the whole length of its body" to its superclass constructor.

• SockeyeSalmon is public, extends Fish, and declares a public noargument constructor that passes "sockeyesalmon" and "bright red with a green head" to its superclass constructor.

Note  For brevity, I have omitted from the Animal hierarchy abstract Robin, Canary, Trout, and Salmon classes that generalize robins, canaries, trout, and salmon. Perhaps you might want to include these classes in the hierarchy.

Although this exercise illustrates the accurate modeling of a natural scenario using inheritance, it also reveals the potential for class explosion—too many classes may be introduced to model a scenario, and it might be difficult to maintain all of these classes. Keep this in mind when modeling with inheritance.

50. Continuing from the previous exercise, declare an Animals class with a main() method. This method first declares an animals array that is initialized to AmericanRobin, RainbowTrout, DomesticCanary, and SockeyeSalmon objects. The method then iterates over this array, first outputting animals[i] (which causes toString() to be called) and then calling each object's eat() and move() methods (demonstrating subtype polymorphism).

51. Continuing from the previous exercise, declare a public Countable interface with a String getID() method. Modify Animal to implement Countable and have this method return kind's value. Modify Animals to initialize the animals array to AmericanRobin, RainbowTrout, DomesticCanary, SockeyeSalmon, RainbowTrout, and AmericanRobin objects. Also, introduce code that computes a census of each kind of animal. This code will use the Census class that is declared in Listing 4-30.

Listing 4-30. The Census Class Stores Census Data on Four Kinds of Animals

public class Census
{
    public final static int SIZE = 4;
    private String[] IDs;
    private int[] counts;

    public String getID()
    { return kind; }

    public static void main()
    { ... }
public Census()
{
    IDs = new String[SIZE];
    counts = new int[SIZE];
}

public String get(int index)
{
    return IDs[index] + " " + counts[index];
}

public void update(String ID)
{
    for (int i = 0; i < IDs.length; i++)
    {
        // If ID not already stored in the IDs array (which is indicated by
        // the first null entry that is found), store ID in this array, and
        // also assign 1 to the associated element in the counts array, to
        // initialize the census for that ID.
        if (IDs[i] == null)
        {
            IDs[i] = ID;
            counts[i] = 1;
            return;
        }
        // If a matching ID is found, increment the associated element in
        // the counts array to update the census for that ID.
        if (IDs[i].equals(ID))
        {
            counts[i]++;
            return;
        }
    }
}

Summary

Inheritance is a hierarchical relationship between similar entity categories in which one category
inherits state and behaviors from at least one other entity category. Inheriting from a single category
is called single inheritance, and inheriting from at least two categories is called multiple inheritance.
Java supports single inheritance and multiple inheritance to facilitate code reuse—why reinvent the wheel? Java supports single inheritance in a class context (via reserved word `extends`), in which a class inherits fields and methods from another class through class extension. Because classes are involved, Java refers to this kind of inheritance as implementation inheritance. Java supports multiple inheritance only in an interface context, in which a class inherits method templates from one or more interfaces through interface implementation (via reserved word `implements`), or in which an interface inherits method templates from one or more interfaces through interface extension (via reserved word `extends`). Because interfaces are involved, Java refers to this kind of inheritance as interface inheritance.

Some real-world entities have the ability to change their forms. The ability to change form is known as polymorphism and is useful to model in a programming language. Although Java supports the coercion, overloading, parametric, and subtype kinds of polymorphism, in this chapter I focused only on subtype polymorphism, which is achieved through upcasting and method overriding.

Every class $X$ exposes an interface (a protocol consisting of constructors, methods, and [possibly] fields that are made available to objects created from other classes for use in creating and communicating with $X$'s objects). Java formalizes the interface concept by providing reserved word `interface`, which is used to introduce a type without implementation.

Although many believe that the interfaces language feature was created as a workaround to Java's lack of support for multiple implementation inheritance, this is not the real reason for its existence. Instead, Java’s interfaces feature was created to give developers the utmost flexibility in designing their applications by decoupling interface from implementation. You should always code to the interface.

Chapter 5 continues to explore the Java language by focusing on nested types, packages, static imports, and exceptions.
Mastering Advanced Language Features, Part 1

In Chapters 2 through 4, I laid a foundation for learning the Java language. In Chapter 5, I will add to this foundation by introducing you to some of Java’s more advanced language features, specifically those features related to nested types, packages, static imports, and exceptions. Additional advanced language features are covered in Chapter 6.

Mastering Nested Types

Classes that are declared outside of any class are known as *top-level classes*. Java also supports *nested classes*, which are classes that are declared as members of other classes or scopes. Nested classes help you implement top-level class architecture.

There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. The latter three categories are known as *inner classes*.

In this section, I will introduce you to static member classes and inner classes. For each kind of nested class, I will provide a brief introduction, an abstract example, and a more practical example. I will then briefly examine the topics of inner classes and memory leaks as well as nesting interfaces within classes and nesting classes within interfaces.

Static Member Classes

A *static member class* is a static member of an enclosing class. Although enclosed, it doesn’t have an enclosing instance of that class and cannot access the enclosing class’s instance fields and invoke its instance methods. However, it can access the enclosing class’s static fields and invoke its static methods, even those members that are declared *private*. Listing 5-1 presents a static member class declaration.
Listing 5-1. Declaring a Static Member Class

class EnclosingClass
{
    private static int i;

    private static void m1()
    {
        System.out.println(i);
    }

    static void m2()
    {
        EnclosedClass.accessEnclosingClass();
    }

    static class EnclosedClass
    {
        static void accessEnclosingClass()
        {
            i = 1;
            m1();
        }

        void accessEnclosingClass2()
        {
            m2();
        }
    }
}

Listing 5-1 declares a top-level class named EnclosingClass with class field i, class methods m1() and m2(), and static member class EnclosedClass. Also, EnclosedClass declares class method accessEnclosingClass() and instance method accessEnclosingClass2().

Because accessEnclosingClass() is declared static, m2() must be prefixed with EnclosedClass and the member access operator to call this method.

Listing 5-2 presents the source code to an application class that demonstrates how to invoke EnclosedClass's accessEnclosingClass() class method and instantiate EnclosedClass and invoke its accessEnclosingClass2() instance method.

Listing 5-2. Invoking a Static Member Class’s Class and Instance Methods

public class SMCDemo
{
    public static void main(String[] args)
    {
        EnclosingClass.EnclosedClass.accessEnclosingClass(); // Output: 1
        EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass();
        ec.accessEnclosingClass2(); // Output: 1
    }
}
Listing 5-2’s main() method reveals that you must prefix the name of an enclosed class with the name of its enclosing class to invoke a class method, for example, EnclosingClass.EnclosedClass.accessEnclosingClass();.

This listing also reveals that you must prefix the name of the enclosed class with the name of its enclosing class when instantiating the enclosed class, for example, EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass(). You can then invoke the instance method in the normal manner, for example, ec.accessEnclosingClass2();.

Static member classes have their uses. For example, Listing 5-3’s Double and Float static member classes provide different implementations of their enclosing Rectangle class. The Float version occupies less memory because of its 32-bit float fields, and the Double version provides greater accuracy because of its 64-bit double fields.

Listing 5-3. Using Static Member Classes to Declare Multiple Implementations of Their Enclosing Class

```java
abstract class Rectangle
{
    abstract double getX();
    abstract double getY();
    abstract double getWidth();
    abstract double getHeight();

    static class Double extends Rectangle
    {
        private double x, y, width, height;

        Double(double x, double y, double width, double height)
        {
            this.x = x;
            this.y = y;
            this.width = width;
            this.height = height;
        }

        double getX() { return x; }
        double getY() { return y; }
        double getWidth() { return width; }
        double getHeight() { return height; }
    }

    static class Float extends Rectangle
    {
        private float x, y, width, height;

        Float(float x, float y, float width, float height)
        {
            this.x = x;
            this.y = y;
            this.width = width;
            this.height = height;
        }
    }

    Double doubleBox()
    {
        return new Double(0.0, 0.0, 1000.0, 2000.0);
    }

    Float floatBox()
    {
        return new Float(0.0f, 0.0f, 1000.0f, 2000.0f);
    }
}
```
double getX() { return x; }
double getY() { return y; }
double getWidth() { return width; }
double getHeight() { return height; }
}

// Prevent subclassing. Use the type-specific Double and Float
// implementation subclass classes to instantiate.
private Rectangle() {};

boolean contains(double x, double y)
{
    return (x >= getX() && x < getX() + getWidth()) &&
    (y >= getY() && y < getY() + getHeight());
}
}

Listing 5-3’s Rectangle class demonstrates nested subclasses. Each of the Double and Float static
member classes subclass the abstract Rectangle class, providing private floating-point or double
precision floating-point fields and overriding Rectangle’s abstract methods to return these fields’
values as doubles.

Rectangle is abstract because it makes no sense to instantiate this class. Because it also makes
no sense to directly extend Rectangle with new implementations (the Double and Float nested
subclasses should be sufficient), its default constructor is declared private. Instead, you must
instantiate Rectangle.Float (to save memory) or Rectangle.Double (when accuracy is required),
as demonstrated by Listing 5-4.

Listing 5-4. Creating and Using Different Rectangle Implementations

public class SMCDemo
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle.Double(10.0, 10.0, 20.0, 30.0);
        System.out.println("x = " + r.getX());
        System.out.println("y = " + r.getY());
        System.out.println("width = " + r.getWidth());
        System.out.println("height = " + r.getHeight());
        System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
        System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
        System.out.println();
        r = new Rectangle.Float(10.0f, 10.0f, 20.0f, 30.0f);
        System.out.println("x = " + r.getX());
        System.out.println("y = " + r.getY());
        System.out.println("width = " + r.getWidth());
        System.out.println("height = " + r.getHeight());
        System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
        System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
    }
}
Listing 5-4 first instantiates Rectangle's Double subclass via new Rectangle.Double(10.0, 10.0, 20.0, 30.0) and then invokes its various methods. Continuing, Listing 5-4 instantiates Rectangle's Float subclass via new Rectangle.Float(10.0f, 10.0f, 20.0f, 30.0f) before invoking Rectangle methods on this instance.

Compile both listings (javac SMCDemo.java or javac *.java) and run the application (java SMCDemo). You will then observe the following output:

```
x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false
```

```
x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false
```

Java’s class library contains many static member classes. For example, the java.lang.Character class encloses a static member class named Subset whose instances represent subsets of the Unicode character set. Additional examples include java.util.AbstractMap.SimpleEntry and java.io.ObjectInputStream.GetField.

**Note** When you compile an enclosing class that contains a static member class, the compiler creates a classfile for the static member class whose name consists of its enclosing class's name, a dollar-sign character, and the static member class’s name. For example, compile Listing 5-1 and you will discover EnclosingClass$EnclosedClass.class in addition to EnclosingClass.class. This format also applies to nonstatic member classes.

**Nonstatic Member Classes**

A nonstatic member class is a non-static member of an enclosing class. Each instance of the nonstatic member class implicitly associates with an instance of the enclosing class. The nonstatic member class's instance methods can call instance methods in the enclosing class and access the enclosing class instance's nonstatic fields. Listing 5-5 presents a nonstatic member class declaration.
Listing 5-5. Declaring a Nonstatic Member Class

class EnclosingClass
{
    private int i;

    private void m()
    {
        System.out.println(i);
    }

class EnclosedClass
{
    void accessEnclosingClass()
    {
        i = 1;
        m();
    }
}

Listing 5-5 declares a top-level class named EnclosingClass with instance field i, instance method m1(), and nonstatic member class EnclosedClass. Furthermore, EnclosedClass declares instance method accessEnclosingClass().

Because accessEnclosingClass() is nonstatic, EnclosedClass must be instantiated before this method can be called. This instantiation must take place via an instance of EnclosingClass. Listing 5-6 accomplishes these tasks.

Listing 5-6. Calling a Nonstatic Member Class’s Instance Method

public class NSMCDemo
{
    public static void main(String[] args)
    {
        EnclosingClass ec = new EnclosingClass();
        ec.new EnclosedClass().accessEnclosingClass(); // Output: 1
    }
}

Listing 5-6’s main() method first instantiates EnclosingClass and saves its reference in local variable ec. Then, main() uses this reference as a prefix to the new operator to instantiate EnclosedClass, whose reference is then used to call accessEnclosingClass(), which outputs 1.

Note Prefixing new with a reference to the enclosing class is rare. Instead, you will typically call an enclosed class’s constructor from within a constructor or an instance method of its enclosing class.
Suppose you need to maintain a to-do list of items, where each item consists of a name and a description. After some thought, you create Listing 5-7’s ToDo class to implement these items.

Listing 5-7. Implementing To-Do Items as Name-Description Pairs

class ToDo
{
    private String name;
    private String desc;

    ToDo(String name, String desc)
    {
        this.name = name;
        this.desc = desc;
    }

    String getName()
    {
        return name;
    }

    String getDesc()
    {
        return desc;
    }

    @Override
    public String toString()
    {
        return "Name = " + getName() + ", Desc = " + getDesc();
    }
}

You next create a ToDoList class to store ToDo instances. ToDoList uses its ToDoArray nonstatic member class to store ToDo instances in a growable array; you don’t know how many instances will be stored, and Java arrays have fixed lengths. See Listing 5-8.

Listing 5-8. Storing a Maximum of Two ToDo Instances in a ToDoArray Instance

class ToDoList
{
    private ToDoArray todoArray;
    private int index = 0;

    ToDoList()
    {
        todoArray = new ToDoArray(2);
    }
boolean hasMoreElements()
{
    return index < toDoArray.size();
}

ToDo nextElement()
{
    return toDoArray.get(index++);
}

void add(ToDo item)
{
    toDoArray.add(item);
}

private class ToDoArray
{
    private ToDo[] toDoArray;
    private int index = 0;

    ToDoArray(int initSize)
    {
        toDoArray = new ToDo[initSize];
    }

    void add(ToDo item)
    {
        if (index >= toDoArray.length)
        {
            ToDo[] temp = new ToDo[toDoArray.length*2];
            for (int i = 0; i < toDoArray.length; i++)
                temp[i] = toDoArray[i];
            toDoArray = temp;
        }
        toDoArray[index++] = item;
    }

    ToDo get(int i)
    {
        return toDoArray[i];
    }

    int size()
    {
        return index;
    }
}

As well as providing an add() method to store ToDo instances in the ToDoArray instance, ToDoList provides hasMoreElements() and nextElement() methods to iterate over and return the stored instances. Listing 5-9 demonstrates these methods.
Listing 5-9. Creating and Iterating Over a ToDoList of ToDo Instances

```java
public class NSMCDemo
{
    public static void main(String[] args)
    {
        ToDoList toDoList = new ToDoList();
        toDoList.add(new ToDo("#1", "Do laundry.");
        toDoList.add(new ToDo("#2", "Buy groceries.");
        toDoList.add(new ToDo("#3", "Vacuum apartment.");
        toDoList.add(new ToDo("#4", "Write report.");
        toDoList.add(new ToDo("#5", "Wash car.");
        while (toDoList.hasMoreElements())
            System.out.println(toDoList.nextElement());
    }
}
```

Compile all three listings (javac NSMCDemo.java or javac *.java) and run the application (java NSMCDemo). You will then observe the following output:

Name = #1, Desc = Do laundry.
Name = #2, Desc = Buy groceries.
Name = #3, Desc = Vacuum apartment.
Name = #4, Desc = Write report.
Name = #5, Desc = Wash car.

Java's class library presents many examples of nonstatic member classes. For example, the java.util package's HashMap class declares private HashIterator, ValueIterator, KeyIterator, and EntryIterator classes for iterating over a hashmap's values, keys, and entries. (I will discuss HashMap in Chapter 9.)

**Note** Code within an enclosed class can obtain a reference to its enclosing class instance by qualifying reserved word this with the enclosing class's name and the member access operator. For example, if code within accessEnclosingClass() needed to obtain a reference to its EnclosingClass instance, it would specify EnclosingClass.this.

### Anonymous Classes

An anonymous class is a class without a name. Furthermore, it is not a member of its enclosing class. Instead, an anonymous class is simultaneously declared (as an anonymous extension of a class or as an anonymous implementation of an interface) and instantiated any place where it is legal to specify an expression. Listing 5-10 demonstrates an anonymous class declaration and instantiation.
Listing 5-10. Declaring and Instantiating an Anonymous Class That Extends a Class

abstract class Speaker
{
    abstract void speak();
}

public class ACDEmo
{
    public static void main(final String[] args)
    {
        new Speaker()
        {
            String msg = (args.length == 1) ? args[0] : "nothing to say";

            @Override
            void speak()
            {
                System.out.println(msg);
            }
        } .speak();
    }
}

Listing 5-10 introduces an abstract class named Speaker and a concrete class named ACDEmo. The latter class’s main() method declares an anonymous class that extends Speaker and overrides its speak() method. When this method is called, it outputs main()’s first command-line argument or a default message when there are no arguments.

An anonymous class doesn’t have a constructor (because the anonymous class doesn’t have a name). However, its classfile does contain an <init>() method that performs instance initialization. This method calls the superclass’s noargument constructor (prior to any other initialization), which is the reason for specifying Speaker() after new.

Anonymous class instances should be able to access the surrounding scope’s local variables and parameters. However, an instance might outlive the method in which it was conceived (as a result of storing the instance’s reference in a field), and try to access local variables and parameters that no longer exist after the method returns.

Because Java cannot allow this illegal access, which would most likely crash the virtual machine, it lets an anonymous class instance only access local variables and parameters that are declared final (see Listing 5-10). On encountering a final local variable/parameter name in an anonymous class instance, the compiler does one of two things:

- If the variable’s type is primitive (int or double, for example), the compiler replaces its name with the variable’s read-only value.
- If the variable’s type is reference (String, for example), the compiler introduces, into the classfile, a synthetic variable (a manufactured variable) and code that stores the local variable’s/parameter’s reference in the synthetic variable.
Listing 5-11 demonstrates an alternative anonymous class declaration and instantiation.

Listing 5-11. Declaring and Instantiating an Anonymous Class That Implements an Interface

```java
interface Speakable
{
    void speak();
}

public class ACDemo
{
    public static void main(final String[] args)
    {
        new Speakable()
        {
            String msg = (args.length == 1) ? args[0] : "nothing to say";

            @Override
            public void speak()
            {
                System.out.println(msg);
            }
        }.speak();
    }
}
```

Listing 5-11 is very similar to Listing 5-10. However, instead of subclassing a Speaker class, this listing’s anonymous class implements an interface named Speakable. Apart from the <init>() method calling java.lang.Object() (interfaces have no constructors), Listing 5-11 behaves like Listing 5-10.

Although an anonymous class doesn’t have a constructor, you can provide an instance initializer to handle complex initialization. For example, `new Office() {{addEmployee(new Employee("John Doe"));}}` instantiates an anonymous subclass of Office and adds one Employee object to this instance by calling Office’s addEmployee() method.

You will often find yourself creating and instantiating anonymous classes for their convenience. For example, suppose you need to return a list of all filenames having the .java suffix. The following example shows you how an anonymous class simplifies using the java.io package’s File and FilenameFilter classes to achieve this objective:

```java
String[] list = new File(directory).list(new FilenameFilter()
{
    @Override
    public boolean accept(File f, String s)
    {
        return s.endsWith(".java");
    }
});
```

However, keep in mind that there is a downside to using anonymous classes. Because they are anonymous, you cannot reuse anonymous classes in other parts of your applications.
Local Classes

A local class is a class that is declared anywhere that a local variable is declared. Furthermore, it has the same scope as a local variable. Unlike an anonymous class, a local class has a name and can be reused. Like anonymous classes, local classes only have enclosing instances when used in nonstatic contexts.

A local class instance can access the surrounding scope’s local variables and parameters. However, the local variables and parameters that are accessed must be declared final. For example, Listing 5-12’s local class declaration accesses a final parameter and a final local variable.

Listing 5-12. Declaring a Local Class

class EnclosingClass
{
    void m(final int x)
    {
        final int y = x * 2;
        class LocalClass
        {
            int a = x;
            int b = y;
        }
        LocalClass lc = new LocalClass();
        System.out.println(lc.a);
        System.out.println(lc.b);
    }
}

Listing 5-12 declares EnclosingClass with its instance method m() declaring a local class named LocalClass. This local class declares a pair of instance fields (a and b) that are initialized to the values of final parameter x and final local variable y when LocalClass is instantiated: new EnclosingClass().m(10);, for example. Listing 5-13 demonstrates this local class.

Listing 5-13. Demonstrating a Local Class

public class LCDemo
{
    public static void main(String[] args)
    {
        EnclosingClass ec = new EnclosingClass();
        ec.m(10);
    }
}

After instantiating EnclosingClass, Listing 5-13’s main() method invokes m(10). The called m() method multiplies this argument by 2, instantiates LocalClass, whose <init>() method assigns the argument and the doubled value to its pair of instance fields (in lieu of using a constructor to perform this task); and outputs the LocalClass instance fields. The following output results:

10
20
Local classes help improve code clarity because they can be moved closer to where they are needed. For example, Listing 5-14 declares an Iterator interface and a ToDoList class whose iterator() method returns an instance of its local Iter class as an Iterator instance (because Iter implements Iterator).

Listing 5-14. The Iterator Interface and the ToDoList Class

```java
interface Iterator
{
    boolean hasMoreElements();
    Object nextElement();
}

class ToDoList
{
    private ToDo[] toDoList;
    private int index = 0;

    ToDoList(int size)
    {
        toDoList = new ToDo[size];
    }

    Iterator iterator()
    {
        class Iter implements Iterator
        {
            int index = 0;

            @Override
            public boolean hasMoreElements()
            {
                return index < toDoList.length;
            }

            @Override
            public Object nextElement()
            {
                return toDoList[index++];
            }
        }
        return new Iter();
    }

    void add(ToDo item)
    {
        toDoList[index++] = item;
    }
}
```

Listing 5-15 demonstrates Iterator, the refactored ToDoList class, and Listing 5-7’s ToDo class.
Listing 5-15. Creating and Iterating Over a ToDoList of ToDo Instances with a Reusable Iterator

```java
public class LCDemo {
    public static void main(String[] args) {
        ToDoList toDoList = new ToDoList(5);
        toDoList.add(new ToDo("#1", "Do laundry."));
        toDoList.add(new ToDo("#2", "Buy groceries."));
        toDoList.add(new ToDo("#3", "Vacuum apartment."));
        toDoList.add(new ToDo("#4", "Write report."));
        toDoList.add(new ToDo("#5", "Wash car."));
        Iterator iter = toDoList.iterator();
        while (iter.hasMoreElements())
            System.out.println(iter.nextElement());
    }
}
```

The Iterator instance that is returned from `iterator()` returns ToDo items in the same order as when they were added to the list. Although you can only use the returned Iterator object once, you can call `iterator()` whenever you need a new Iterator object. This capability is a big improvement over the one-shot iterator presented in Listing 5-9.

**Inner Classes and Memory Leaks**

Instances of inner classes contain implicit references to their outer classes. Prolonging the existence of an inner class instance (such as storing its reference in a static variable) can result in a memory leak in which the outer instance may be referencing a large graph of objects that cannot be garbage collected because of the prolonged inner class reference. Consider Listing 5-16’s example.

Listing 5-16. Demonstrating a Memory Leak in the Context of a Local Class

```java
public class InnerLeakDemo {
    public static void main(String[] args) {
        class Outer {
            String s = "outer string";
        }
        class Inner {
            String s = "inner string";
            void print()
                {
                System.out.println(s);
                System.out.println(Outer.this.s);
            }
        }
    }
}
```
Outer o = new Outer();
    Outer.Inner oi = o.new Inner();
    oi.print();

Listing 5-16’s `main()` method declares a local class named `Outer`, which declares a nonstatic member class named `Inner`. Each of `Outer` and `Inner` declares a `String` instance field named `s`, and `Inner` also declares a `print()` method.

After declaring `Outer`, `main()` instantiates this local class and then instantiates its nested `Inner` member. Finally, it invokes `Inner`’s `print()` method.

Compile Listing 5-16 (`javac InnerLeakDemo.java`) and run this application (`java InnerLeakDemo`). You should observe the following output:

```
inner string
outer string
```

This output reveals that `Outer`’s `s` field is present for as long as the reference to its `Inner` class exists. This is an example of a memory leak.

Although this example is trivial, you’ll find it more profoundly demonstrated in an Android context, in which the inner class can outlive its activity outer class and delay the activity from being garbage collected and its resources released. To learn more about this problem, check out Alex Lockwood’s “How to Leak a Context: Handlers & Inner Classes” blog post (www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html).

**Interfaces within Classes and Classes within Interfaces**

Interfaces can be nested within classes. Once declared, an interface is considered to be static even when it is not declared static. For example, Listing 5-17 declares an enclosing class named `X` along with two nested static interfaces named `A` and `B`.

Listing 5-17. Declaring a Pair of Interfaces Within a Class

```java
class X
{
    interface A
    {
    
    }

    static interface B
    {
    
    }
}
```

You would access Listing 5-17’s interfaces in the same way. For example, you would specify `class C implements X.A {}` or `class D implements X.B {}`. 
As with nested classes, nested interfaces help to implement top-level class architecture by being implemented by nested classes. Collectively, these types are nested because they cannot (as in Listing 5-14’s Iter local class) or need not appear at the same level as a top-level class and pollute its package namespace. The java.util.Map.Entry interface and HashMap class is a good example.

Classes can be nested within interfaces. Once declared, a class is considered to be static even when it is not declared static. Although nowhere near as common as nesting an interface within a class, nested classes have a potential use, which Listing 5-18 demonstrates.

Listing 5-18. Tightly Binding a Class to an Interface

```java
public interface Addressable
{
    class Address
    {
        private String boxNumber;
        private String street;
        private String city;

        public Address(String boxNumber, String street, String city)
        {
            this.boxNumber = boxNumber;
            this.street = street;
            this.city = city;
        }

        public String getBoxNumber()
        {
            return boxNumber;
        }

        public String getStreet()
        {
            return street;
        }

        public String getCity()
        {
            return city;
        }

        @Override
        public String toString()
        {
            return boxNumber+" - "+street+" - "+city;
        }
    }

    Address getAddress();
}
```
Listing 5-18 declares an Addressable interface that describes any entity associated with an address, such as a letter, parcel, or postcard. This interface declares an Address class to store the address components, and an Address getAddress() method that returns the address.

Assuming the existence of Letter, Parcel, and Postcard classes whose constructors take Address arguments, the following code fragment shows you how you could construct an array of addressables and then iterate over it, obtaining and printing each address:

```java
Addressable[] addressables =
{
    new Letter(new Addressable.Address("10", "AnyStreet", "AnyTown")),
    new Parcel(new Addressable.Address("20", "Doe Street", "NewTown")),
    new Postcard(new Addressable.Address("30", "Ender Avenue", "AnyCity"))
};

for (int i = 0; i < addressables.length; i++)
    System.out.println(addressables[i].getAddress());
```

Notice that you must specify Addressable.Address to access the nested Address class. (You can use the static imports feature discussed in Chapter 6 to save yourself from typing the Addressable. prefix.)

Why nest Address instead of making it a separate top-level class? The idea here is that there exists a tight relationship between Addressable and Address, and you want to capture this relationship. Also, you might have a different top-level Address class and want to prevent a name conflict.

Nesting a class in an interface is considered by many to be a bad practice, because it goes against the ideas of object-oriented programming and the notion of an interface. However, you should be aware of this capability because you may encounter it in practice.

**Mastering Packages**

Hierarchical structures organize items in terms of hierarchical relationships that exist between those items. For example, a filesystem might contain a taxes directory with multiple year subdirectories, where each subdirectory contains tax information pertinent to that year. Also, an enclosing class might contain multiple nested classes that only make sense in the context of the enclosing class.

Hierarchical structures also help to avoid name conflicts. For example, two files cannot have the same name in a nonhierarchical filesystem (which consists of a single directory). In contrast, a hierarchical filesystem lets same-named files exist in different directories. Similarly, two enclosing classes can contain same-named nested classes. Name conflicts don’t exist because items are partitioned into different namespaces.

Java also supports the partitioning of top-level user-defined types into multiple namespaces to better organize these types and to also prevent name conflicts. Java uses packages to accomplish these tasks.

In this section, I will introduce you to packages. After defining this term and explaining why package names must be unique, I will present the package and import statements. I will next explain how the virtual machine searches for packages and types and then I will present an example that shows you how to work with packages. I will close this section by showing you how to encapsulate a package of classfiles into JAR files.
Tip  Except for the most trivial of top-level types and (typically) those classes that serve as application entry points (they have `main()` methods), you should consider storing your types (especially when they are reusable) in packages. Get into the habit now because you’ll use packages extensively when developing Android apps. Each Android app must be stored in its own unique package.

What Are Packages?

A package is a unique namespace that can contain a combination of top-level classes, other top-level types, and subpackages. Only types that are declared public can be accessed from outside the package. Furthermore, the constants, constructors, methods, and nested types that describe a class’s interface must be declared public to be accessible from beyond the package.

Every package has a name, which must be a nonreserved identifier. The member access operator separates a package name from a subpackage name and separates a package or subpackage name from a type name. For example, the two member access operators in `graphics.shapes.Circle` separate package name `graphics` from the `shapes` subpackage name and separate subpackage name `shapes` from the `Circle` type name.

Note  Each of Oracle’s and Google Android’s standard class libraries organizes its many classes and other top-level types into multiple packages. Many of these packages are subpackages of the standard `java` package. Examples include `java.io` (types related to input/output operations), `java.lang` (language-oriented types), `java.net` (network-oriented types), and `java.util` (utility types).

Package Names Must Be Unique

Suppose you have two different `graphics.shapes` packages, and suppose that each shapes subpackage contains a Circle class with a different interface. When the compiler encounters `System.out.println(new Circle(10.0, 20.0, 30.0).area());` in the source code, it needs to verify that the `area()` method exists.

The compiler will search all accessible packages until it finds a `graphics.shapes` package that contains a Circle class. If the found package contains the appropriate Circle class with an `area()` method, everything is fine. Otherwise, if the Circle class doesn’t have an `area()` method, the compiler will report an error.

This scenario illustrates the importance of choosing unique package names. Specifically, the top-level package name must be unique. The convention in choosing this name is to take your Internet domain name and reverse it. For example, I would choose `ca.tutortutor` as my top-level package name because `tutortutor.ca` is my domain name. I would then specify `ca.tutortutor.graphics.shapes.Circle` to access Circle.
Note  Reversed Internet domain names are not always valid package names. One or more of its component names might start with a digit (6.com), contain a hyphen (-) or other illegal character (aq-x.com), or be one of Java's reserved words (int.com). Convention dictates that you prefix the digit with an underscore (com._6), replace the illegal character with an underscore (com.aq_x), and suffix the reserved word with an underscore (com.int_).

The Package Statement

The package statement identifies the package in which a source file's types are located. This statement consists of reserved word package, followed by a member access operator-separated list of package and subpackage names, followed by a semicolon.

For example, package graphics; specifies that the source file's types locate in a package named graphics, and package graphics.shapes; specifies that the source file's types locate in the graphics package's shapes subpackage.

By convention, a package name is expressed in lowercase. When the name consists of multiple words, each word except for the first word is capitalized.

Only one package statement can appear in a source file. When it is present, nothing apart from comments must precede this statement.

Caution  Specifying multiple package statements in a source file or placing anything apart from comments above a package statement causes the compiler to report an error.

Java implementations map package and subpackage names to same-named directories. For example, an implementation would map graphics to a directory named graphics and would map graphics.shapes to a shapes subdirectory of graphics. The Java compiler stores the classfiles that implement the package's types in the corresponding directory.

Note  When a source file doesn't contain a package statement, the source file's types are said to belong to the unnamed package. This package corresponds to the current directory.

The Import Statement

Imagine having to repeatedly specify ca.tutortutor.graphics.shapes.Circle or some other lengthy package-qualified type name for each occurrence of that type in source code. Java provides an alternative that lets you avoid having to specify package details. This alternative is the import statement.
The import statement imports types from a package by telling the compiler where to look for unqualified type names during compilation. This statement consists of reserved word `import`, followed by a member access operator-separated list of package and subpackage names, followed by a type name or `*` (asterisk), followed by a semicolon.

The `*` symbol is a wildcard that represents all unqualified type names. It tells the compiler to look for such names in the import statement’s specified package, unless the type name is found in a previously searched package. (Using the wildcard doesn’t have a performance penalty or lead to code bloat but can lead to name conflicts, as you will see.)

For example, `import ca.tutortutor.graphics.shapes.Circle;` tells the compiler that an unqualified Circle class exists in the `ca.tutortutor.graphics.shapes` package. Similarly, `import ca.tutortutor.graphics.shapes.*;` tells the compiler to look in this package when it encounters a Circle class, a Rectangle class, or even an Employee class (if Employee hasn’t already been found).

**Tip**  
You should avoid using the `*` wildcard so that other developers can easily see which types are used in source code.

Because Java is case sensitive, package and subpackage names specified in an import statement must be expressed in the same case as that used in the package statement.

When import statements are present in source code, only a package statement and comments can precede them.

**Caution**  
Placing anything other than a package statement, import statements, static import statements (discussed shortly), and comments above an import statement causes the compiler to report an error.

You can run into name conflicts when using the wildcard version of the import statement because any unqualified type name matches the wildcard. For example, you have `graphics.shapes` and `geometry` packages that each contain a Circle class, the source code begins with `import geometry.*;` and `import graphics.shape.*;` statements, and it also contains an unqualified occurrence of Circle. Because the compiler doesn’t know if Circle refers to geometry’s Circle class or graphics.shape’s Circle class, it reports an error. You can fix this problem by qualifying Circle with the correct package name.

**Note**  
The compiler automatically imports the String class and other types from the `java.lang` package, which is why it’s not necessary to qualify String with `java.lang`. 
Searching for Packages and Types

Newcomers to Java who first start to work with packages often become frustrated by “no class definition found” and other errors. This frustration can be partly avoided by understanding how the virtual machine searches for packages and types.

This section explains how the search process works. To understand this process, you need to realize that the compiler is a special Java application that runs under the control of the virtual machine. Furthermore, there are two different forms of search.

Compile-Time Search

When the compiler encounters a type expression (such as a method call) in source code, it must locate that type’s declaration to verify that the expression is legal (a method exists in the type’s class whose parameter types match the types of the arguments passed in the method call, for example).

The compiler first searches the Java platform packages (which contain class library types). It then searches extension packages (for extension types). If the `-sourcepath` command-line option is specified when starting the virtual machine (via `javac`), the compiler searches the indicated path’s source files.

Note: Java platform packages are stored in `rt.jar` and a few other important JAR files. Extension packages are stored in a special extensions directory named `ext`.

Otherwise, the compiler searches the user classpath (in left-to-right order) for the first user classfile or source file containing the type. If no user classpath is present, the current directory is searched. If no package matches or the type still cannot be found, the compiler reports an error. Otherwise, the compiler records the package information in the classfile.

Note: The user classpath is specified via the `-classpath` (or `-cp`) option used to start the virtual machine or, when not present, the `CLASSPATH` environment variable.

Runtime Search

When the compiler or any other Java application runs, the virtual machine will encounter types and must load their associated classfiles via special code known as a classloader. The virtual machine will use the previously stored package information that is associated with the encountered type in a search for that type’s classfile.
The virtual machine searches the Java platform packages, followed by extension packages, followed by the user classpath (in left-to-right order) for the first classfile that contains the type. If no user classpath is present, the current directory is searched. If no package matches or the type cannot be found, a “no class definition found” error is reported. Otherwise, the classfile is loaded into memory.

Note Whether you use the -classpath/-cp option or the CLASSPATH environment variable to specify a user classpath, there is a specific format that must be followed. Under Windows, this format is expressed as path1;path2;..., where path1, path2, and so on are the locations of package directories. Under Mac OS X, Unix, and Linux, this format changes to path1:path2:....

Playing with Packages

Suppose your application needs to log messages to the console, to a file, or to another destination. It can accomplish this task with the help of a logging library. My implementation of this library consists of an interface named Logger, an abstract class named LoggerFactory, and a pair of package-private classes named Console and File.

Note The logging library that I present is an example of the Abstract Factory design pattern, which is presented on page 87 of Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; ISBN: 0201633612).

Listing 5-19 presents the Logger interface, which describes objects that log messages.

Listing 5-19. Describing Objects That Log Messages via the Logger Interface

```java
package logging;

public interface Logger {
    boolean connect();
    boolean disconnect();
    boolean log(String msg);
}
```

Each of the connect(), disconnect(), and log() methods returns true upon success and false upon failure. (Later in this chapter, you will discover a better technique for dealing with failure.) These methods are not declared public explicitly because an interface’s methods are implicitly public.

Listing 5-20 presents the LoggerFactory abstract class.
Listing 5-20. Obtaining a Logger for Logging Messages to a Specific Destination

```java
package logging;

class LoggerFactory {
    public final static int CONSOLE = 0;
    public final static int FILE = 1;

    public static Logger newLogger(int dstType, String... dstName) {
        switch (dstType) {
            case CONSOLE: return new Console(dstName.length == 0 ? null : dstName[0]);
            case FILE   : return new File(dstName.length == 0 ? null : dstName[0]);
            default     : return null;
        }
        return null;
    }
}
```

newLogger() returns a Logger object for logging messages to an appropriate destination. It uses the varargs (variable arguments) feature (see Chapter 3) to optionally accept an extra String argument for those destination types that require the argument. For example, FILE requires a filename.

Listing 5-21 presents the package-private Console class; this class is not accessible beyond the classes in the logging package because reserved word class is not preceded by reserved word public.

Listing 5-21. Logging Messages to the Console

```java
package logging;

class Console implements Logger {
    private String dstName;

    Console(String dstName) {
        this.dstName = dstName;
    }

    @Override
    public boolean connect() {
        return true;
    }
}
@Override
public boolean disconnect()
{
    return true;
}

@Override
public boolean log(String msg)
{
    System.out.println(msg);
    return true;
}
}

Console’s package-private constructor saves its argument, which most likely will be null because there is no need for a String argument. Perhaps a future version of Console will use this argument to identify one of multiple console windows.

Listing 5-22 presents the package-private File class.

Listing 5-22. Logging Messages to a File (Eventually)

package logging;

class File implements Logger {
    private String dstName;

    File(String dstName)
    {
        this.dstName = dstName;
    }

    @Override
    public boolean connect()
    {
        if (dstName == null)
            return false;
        System.out.println("opening file "+ dstName);
        return true;
    }

    @Override
    public boolean disconnect()
    {
        if (dstName == null)
            return false;
        System.out.println("closing file "+ dstName);
        return true;
    }
@Override
public boolean log(String msg)
{
    if (dstName == null)
        return false;
    System.out.println("writing "+msg+" to file "+ dstName);
    return true;
}

Unlike Console, File requires a nonnull argument. Each method first verifies that this argument is not null. If the argument is null, the method returns false to signify failure. (In Chapter 11, I refactor File to incorporate appropriate file-writing code.)

The logging library allows us to introduce portable logging code into an application. Apart from a call to newLogger(), this code will remain the same regardless of the logging destination. Listing 5-23 presents an application that tests this library.

Listing 5-23. Testing the Logging Library

import logging.Logger;
import logging.LoggerFactory;

public class TestLogger
{
    public static void main(String[] args)
    {
        Logger logger = LoggerFactory.newLogger(LoggerFactory.CONSOLE);
        if (logger.connect())
        {
            logger.log("test message #1");
            logger.disconnect();
        }
        else
        {
            System.out.println("cannot connect to console-based logger");
            logger = LoggerFactory.newLogger(LoggerFactory.FILE, "x.txt");
            if (logger.connect())
            {
                logger.log("test message #2");
                logger.disconnect();
            }
            else
            {
                System.out.println("cannot connect to file-based logger");
                logger = LoggerFactory.newLogger(LoggerFactory.FILE);
                if (logger.connect())
                {
                    logger.log("test message #3");
                    logger.disconnect();
                }
                else
                {
                    System.out.println("cannot connect to file-based logger");
                }
            }
        }
    }
}
Follow the steps (which assume that the JDK has been installed) to create the logging package and TestLogger application, and to run this application.

1. Create a new directory and make this directory current.
2. Create a logging directory in the current directory.
3. Copy Listing 5-19 to a file named Logger.java in the logging directory.
4. Copy Listing 5-20 to a file named LoggerFactory.java in the logging directory.
5. Copy Listing 5-21 to a file named Console.java in the logging directory.
6. Copy Listing 5-22 to a file named File.java in the logging directory.
7. Copy Listing 5-23 to a file named TestLogger.java in the current directory.
8. Execute javac TestLogger.java, which also compiles logger’s source files.

After completing these steps, you should observe the following output from the TestLogger application:

test message #1
opening file x.txt
writing test message #2 to file x.txt
closing file x.txt
cannot connect to file-based logger

What happens when logging is moved to another location? For example, move logging to the root directory and run TestLogger. You will now observe an error message about the virtual machine not finding the logging package and its LoggerFactory classfile.

You can solve this problem by specifying -classpath/-cp when running the java tool or by adding the location of the logging package to the CLASSPATH environment variable. For example, I chose to use -classpath (which I find more convenient) in the following Windows-specific command line:

java -classpath \.; TestLogger

The backslash represents the root directory in Windows. (I could have specified a forward slash as an alternative.) Also, the period represents the current directory. If it is missing, the virtual machine complains about not finding the TestLogger classfile.

Tip If you discover an error message where the virtual machine reports that it cannot find an application classfile, try appending a period character to the classpath. Doing so will probably fix the problem.
Packages and JAR Files

The JDK provides a `jar` tool that is used to archive classfiles in JAR (Java ARchive) files and is also used to extract a JAR file's classfiles. It probably comes as no surprise that you can store packages in JAR files, which greatly simplify the distribution of your package-based class libraries.

To show you how easy it is to store a package in a JAR file, you will create a `logger.jar` file that contains the logging package’s four classfiles (`Logger.class`, `LoggerFactory.class`, `Console.class`, and `File.class`). Complete the following steps to accomplish this task:

1. Make sure that the current directory contains the previously created `logging` directory with its four classfiles.
2. Execute the following command:
   ```
   jar cf logger.jar logging/*.class
   ```
   The `c` option stands for “create new archive” and the `f` option stands for “specify archive filename.”
   You could alternatively execute the following command:
   ```
   jar cf logger.jar logging/*.class
   ```
   You should now find a `logger.jar` file in the current directory. To prove to yourself that this file contains the four classfiles, execute the following command, where the `t` option stands for “list table of contents”:
   ```
   jar tf logger.jar
   ```
   You can run `TestLogger.class` by adding `logger.jar` to the classpath. For example, you can run `TestLogger` under Windows via the following command:
   ```
   java -classpath logger.jar;. TestLogger
   ```

   **Note** Although you can create your own logging framework, doing so is a waste of time. Instead, you should leverage the `java.util.logging` package (see Chapter 16) that’s included in the standard class library.

Mastering Static Imports

An interface should only be used to declare a type. However, some developers violate this principle by using interfaces to only export constants. Such interfaces are known as constant interfaces, and Listing 5-24 presents an example.
Listing 5-24. Declaring a Constant Interface

```java
interface Directions {
    int NORTH = 0;
    int SOUTH = 1;
    int EAST = 2;
    int WEST = 3;
}
```

Developers who resort to constant interfaces do so to avoid having to prefix a constant’s name with the name of its class (as in Math.PI, where PI is a constant in the java.lang.Math class). They do this by implementing the interface (see Listing 5-25).

Listing 5-25. Implementing a Constant Interface

```java
public class TrafficFlow implements Directions {
    public static void main(String[] args) {
        showDirection((int) (Math.random() * 4));
    }

    static void showDirection(int dir) {
        switch (dir) {
            case NORTH: System.out.println("Moving north"); break;
            case SOUTH: System.out.println("Moving south"); break;
            case EAST: System.out.println("Moving east"); break;
            case WEST: System.out.println("Moving west");
        }
    }
}
```

Listing 5-25’s TrafficFlow class implements Directions for the sole purpose of not having to specify Directions.NORTH, Directions.SOUTH, Directions.EAST, and Directions.WEST. This is an appalling misuse of an interface. These constants are nothing more than an implementation detail that should not be allowed to leak into the class’s exported interface because they might confuse the class’s users (what is the purpose of these constants?). Also, they represent a future commitment: even when the class no longer uses these constants, the interface must remain to ensure binary compatibility.

Java 5 introduced an alternative that satisfies the desire for constant interfaces while avoiding their problems. This static imports feature lets you import a class’s static members so that you don’t have to qualify them with their class names. It’s implemented via a small modification to the import statement, as follows:

```java
import static packagepec .classname . (staticmemberofname | * );
```
The static import statement specifies static after import. It then specifies a member access operator-separated list of package and subpackage names, which is followed by the member access operator and a class's name. Once again, the member access operator is specified, followed by a single static member name or the asterisk wildcard.

Caution  Placing anything apart from a package statement, import/static import statements, and comments above a static import statement causes the compiler to report an error.

You specify a single static member name to import only that name.

import static java.lang.Math.PI;  // Import the PI static field only.
import static java.lang.Math.cos; // Import the cos() static method only.

In contrast, you specify the wildcard to import all static member names.

import static java.lang.Math.*;   // Import all static members from Math.

You can now refer to the static member(s) without having to specify the class name.

System.out.println(cos(PI));

Using multiple static import statements can result in name conflicts, which causes the compiler to report errors. For example, suppose your geom package contains a Circle class with a static member named PI. Now suppose you specify import static java.lang.Math.*; and import static geom.Circle.*; at the top of your source file. Finally, suppose you specify System.out.println(PI); somewhere in that file's code. The compiler reports an error because it doesn't know whether PI belongs to Math or to Circle.

Caution  Overuse of static imports can make your code unreadable and unmaintainable. Anyone reading your code could have a hard time finding out which class a static member comes from, especially when importing all static member names from a class. Also, static imports pollute the code's namespace with all of the static members you import. Eventually, you may run into name conflicts that are hard to resolve.

Mastering Exceptions

In an ideal world, nothing bad ever happens when an application runs. For example, a file always exists when the application needs to open the file, the application is always able to connect to a remote computer, and the virtual machine never runs out of memory when the application needs to instantiate objects.
In contrast, real-world applications occasionally attempt to open files that don’t exist, attempt to connect to remote computers that are unable to communicate with them, and require more memory than the virtual machine can provide. Your goal is to write code that properly responds to these and other exceptional situations (exceptions).

This section introduces you to exceptions. After defining this term, I will look at representing exceptions in source code. I will then examine the topics of throwing and handling exceptions and conclude by discussing how to perform cleanup tasks before a method returns, whether or not an exception has been thrown.

**What Are Exceptions?**

An *exception* is a divergence from an application’s normal behavior. For example, the application attempts to open a nonexistent file for reading. The normal behavior is to successfully open the file and begin reading its contents. However, the file cannot be read when the file doesn’t exist.

This example illustrates an exception that cannot be prevented. However, a workaround is possible. For example, the application can detect that the file doesn’t exist and take an alternate course of action, which might include telling the user about the problem. Unpreventable exceptions where workarounds are possible must not be ignored.

Exceptions can occur because of poorly written code. For example, an application might contain code that accesses each element in an array. Because of careless oversight, the array-access code might attempt to access a nonexistent array element, which leads to an exception. This kind of exception is preventable by writing correct code.

Finally, an exception might occur that cannot be prevented and for which there is no workaround. For example, the virtual machine might run out of memory, or perhaps it cannot find a classfile. This kind of exception, known as an *error*, is so serious that it’s impossible (or at least inadvisable) to work around; the application must terminate, presenting a message to the user that explains why it’s terminating.

**Representing Exceptions in Source Code**

An exception can be represented via error codes or objects. After discussing each kind of representation and explaining why objects are superior, I will introduce you to Java's exception and error class hierarchy, emphasizing the difference between checked and runtime exceptions. I will close my discussion on representing exceptions in source code by discussing custom exception classes.

**Error Codes vs. Objects**

One way to represent exceptions in source code is to use error codes. For example, a method might return true on success and false when an exception occurs. Alternatively, a method might return 0 on success and a nonzero integer value that identifies a specific kind of exception.

Developers traditionally designed methods to return error codes; I demonstrated this tradition in each of the three methods in Listing 5-19’s Logger interface. Each method returns true on success or returns false to represent an exception (unable to connect to the logger, for example).
Although a method’s return value must be examined to see if it represents an exception, error codes are all too easy to ignore. For example, a lazy developer might ignore the return code from Logger’s connect() method and attempt to call log(). Ignoring error codes is one reason why a new approach to dealing with exceptions has been invented.

This new approach is based on objects. When an exception occurs, an object representing the exception is created by the code that was running when the exception occurred. Details describing the exception’s surrounding context are stored in the object. These details are later examined to work around the exception.

The object is then thrown or handed off to the virtual machine to search for a handler, code that can handle the exception. (If the exception is an error, the application should not provide a handler because errors are so serious [such as the virtual machine has run out of memory] that there’s practically nothing that can be done about them.) When a handler is located, its code is executed to provide a workaround. Otherwise, the virtual machine terminates the application.

**Caution** Code that handles exceptions can be a source of bugs because it’s often not thoroughly tested. Always make sure to test any code that handles exceptions.

Apart from being too easy to ignore, an error code’s Boolean or integer value is less meaningful than an object name. For example, fileNotFound is self-evident, but what does false mean? Also, an object can contain information about what led to the exception. These details can be helpful to a suitable workaround.

**The Throwable Class Hierarchy**

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in java.lang.Throwable, the ultimate superclass for all throwables (exception and error objects—exceptions and errors, for short—that can be thrown). Figure 5-1 reveals Throwable and its immediate subclasses.

![Figure 5-1. The exceptions hierarchy is rooted in the Throwable class](image)

Exception is the root class for all exception-oriented throwables. Similarly, Error is the root class for all error-oriented throwables.

Table 5-1 identifies and describes most of Throwable’s constructors and methods.
It’s not uncommon for a class’s public methods to call helper methods that throw various exceptions. A public method will probably not document exceptions thrown from a helper method because they are implementation details that often should not be visible to the public method’s caller.

However, because this exception might be helpful in diagnosing the problem, the public method can wrap the lower-level exception in a higher-level exception that is documented in the public method’s contract interface. The wrapped exception is known as a cause because its existence causes the higher-level exception to be thrown.

A cause is created by invoking the Throwable(Throwable cause) or Throwable(String message, Throwable cause) constructor, which invoke the initCause() method to store the cause. If you don’t call either constructor, you can alternatively call initCause() directly, but you must do so immediately after creating the throwable. Call the getCause() method to return the cause.

When an exception is thrown, it leaves behind a stack of unfinished method calls. Throwable’s constructors call fillInStackTrace() to record this stack trace information, which is output by calling printStackTrace().

---

Table 5-1. Throwable’s Constructors and Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throwable()</td>
<td>Create a throwable with a null detail message and cause.</td>
</tr>
<tr>
<td>Throwable(String message)</td>
<td>Create a throwable with the specified detail message and a null cause.</td>
</tr>
<tr>
<td>Throwable(String message, Throwable cause)</td>
<td>Create a throwable with the specified detail message and cause.</td>
</tr>
<tr>
<td>Throwable(Throwable cause)</td>
<td>Create a throwable whose detail message is the string representation of a nonnull cause or null.</td>
</tr>
<tr>
<td>Throwable fillInStackTrace()</td>
<td>Fill in the execution stack trace. This method records information about the current state of the stack frames for the current thread within this throwable. (I discuss threads in Chapter 7.)</td>
</tr>
<tr>
<td>Throwable getCause()</td>
<td>Return the cause of this throwable. When there is no cause, null is returned.</td>
</tr>
<tr>
<td>String getMessage()</td>
<td>Return this throwable’s detail message, which might be null.</td>
</tr>
<tr>
<td>StackTraceElement[] getStackTrace()</td>
<td>Provide programmatic access to the stack trace information printed by printStackTrace() as an array of stack trace elements, each representing one stack frame.</td>
</tr>
<tr>
<td>Throwable initCause(Throwable cause)</td>
<td>Initialize the cause of this throwable to the specified value.</td>
</tr>
<tr>
<td>void printStackTrace()</td>
<td>Print this throwable and its backtrace of stack frames to the standard error stream.</td>
</tr>
<tr>
<td>void setStackTrace(StackTraceElement[] stackTrace)</td>
<td>Set the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.</td>
</tr>
</tbody>
</table>
The `getStackTrace()` method provides programmatic access to the stack trace by returning this information as an array of `java.lang.StackTraceElement` instances—each instance represents one entry. `StackTraceElement` provides methods to return stack trace information. For example, `String getMethodName()` returns the name of an unfinished method.

The `setStackTrace()` method is designed for use by Remote Procedure Call (RPC) frameworks (see http://en.wikipedia.org/wiki/Remote_procedure_call) and other advanced systems, allowing the client to override the default stack trace that is generated by `fillInStackTrace()` when a throwable is constructed or deserialized when a throwable is read from a serialization stream. (I will discuss serialization in Chapter 11.)

Moving down the throwable hierarchy, you encounter the `java.lang.Exception` and `java.lang.Error` classes, which respectively represent exceptions and errors. Each class offers four constructors that pass their arguments to their `Throwable` counterparts but provides no methods apart from those that are inherited from `Throwable`.

Exception is itself subclassed by `java.lang.CloneNotSupportedException` (discussed in Chapter 4), `java.lang.IOException` (discussed in Chapter 11), and other classes. Similarly, Error is itself subclassed by `java.lang.AssertionError` (discussed in Chapter 6), `java.lang.OutOfMemoryError`, and other classes.

---

**Caution** Never instantiate `Throwable`, `Exception`, or `Error`. The resulting objects are meaningless because they are too generic.

---

**Checked Exceptions vs. Runtime Exceptions**

A *checked exception* is an exception that represents a problem with the possibility of recovery and for which the developer must provide a workaround. The compiler checks the code to ensure that the exception is handled in the method where it is thrown, or is explicitly identified as being handled elsewhere.

Exception and all subclasses except for `java.lang.RuntimeException` (and its subclasses) describe checked exceptions. For example, the `CloneNotSupportedException` and `IOException` classes describe checked exceptions. (`CloneNotSupportedException` should not be checked because there is no runtime workaround for this kind of exception.)

A *runtime exception* is an exception that represents a coding mistake. This kind of exception is also known as an *unchecked exception* because it doesn’t need to be handled or explicitly identified—the mistake must be fixed. Because these exceptions can occur in many places, it would be burdensome to be forced to handle them.

`RuntimeException` and its subclasses describe unchecked exceptions. For example, `java.lang.ArithmeticException` describes arithmetic problems such as integer division by zero. Another example is `java.lang.ArrayIndexOutOfBoundsException`, which is thrown when you attempt to access an array element with a negative index or an index that is greater than or equal to the length of the array. (In hindsight, `RuntimeException` should have been named `UncheckedException` because all exceptions occur at runtime.)
Note  Many developers are unhappy with checked exceptions because of the work involved in having to handle them. This problem is made worse by libraries providing methods that throw checked exceptions when they should throw unchecked exceptions. As a result, many modern languages support only unchecked exceptions.

Custom Exception Classes

You can declare your own exception classes. Before doing so, ask yourself if an existing exception class in the standard class library meets your needs. If you find a suitable class, you should reuse it. (Why reinvent the wheel?) Other developers will already be familiar with the existing class, and this knowledge will make your code easier to learn. When no existing class meets your needs, think about whether to subclass Exception or RuntimeException. In other words, will your exception class be checked or unchecked? As a rule of thumb, your class should subclass RuntimeException if you think that it will describe a coding mistake.

Tip  When you name your class, follow the convention of providing an Exception suffix. This suffix clarifies that your class describes an exception.

Suppose you are creating a Media class whose static methods are to perform media-oriented utility tasks. For example, one method converts sound files in non-MP3 media formats to MP3 format. This method will be passed source file and destination file arguments and will convert the source file to the format implied by the destination file’s extension.

Before performing the conversion, the method needs to verify that the source file’s format agrees with the format implied by its file extension. If there is no agreement, an exception must be thrown. Furthermore, this exception must store the expected and existing media formats so that a handler can identify them when presenting a message to the user.

Because Java’s class library doesn’t provide a suitable exception class, you decide to introduce a class named InvalidMediaFormatException. Detecting an invalid media format is not the result of a coding mistake, and so you also decide to extend Exception to indicate that the exception is checked. Listing 5-26 presents this class’s declaration.

Listing 5-26. Declaring a Custom Exception Class

```java
package media;

public class InvalidMediaFormatException extends Exception {
    private String expectedFormat;
    private String existingFormat;
```
InvalidMediaFormatException provides a constructor that calls Exception's public Exception(String message) constructor with a detail message that includes the expected and existing formats. It is wise to capture such details in the detail message because the problem that led to the exception might be hard to reproduce.

InvalidMediaFormatException also provides getExpectedFormat() and getExistingFormat() methods that return these formats. Perhaps a handler will present this information in a message to the user. Unlike the detail message, this message might be localized, expressed in the user's language (French, German, English, and so on).

### Throwing Exceptions

Now that you have created an InvalidMediaFormatException class, you can declare the Media class and begin to code its convert() method. The initial version of this method validates its arguments and then verifies that the source file's media format agrees with the format implied by its file extension. Check out Listing 5-27.

Listing 5-27. Throwing Exceptions from the convert() Method

```java
package media;

import java.io.IOException;

public final class Media {
    public static void convert(String srcName, String dstName)
            throws InvalidMediaFormatException, IOException {
        if (srcName == null)
            throw new NullPointerException(srcName + " is null");
    }
}
```

www.it-ebooks.info
if (dstName == null)
    throw new NullPointerException(dstName + " is null");

// Code to access source file and verify that its format matches the
// format implied by its file extension.

// Assume that the source file’s extension is RM (for Real Media) and
// that the file’s internal signature suggests that its format is
// Microsoft WAVE.
String expectedFormat = "RM";
String existingFormat = "WAVE";
throw new InvalidMediaFormatException(expectedFormat, existingFormat);
}

Listing 5-27 declares the Media class to be final because this utility class will only consist of class
methods and there’s no reason to extend it.

Media’s convert() method appends throws InvalidMediaFormatException, IOException to its header. A throws clause identifies all checked exceptions that are thrown out of the method and must be handled by some other method. It consists of reserved word throws followed by a comma-separated list of checked exception class names and is always appended to a method header. The convert() method’s throws clause indicates that this method is capable of throwing an InvalidMediaException or IOException instance to the virtual machine.

convert() also demonstrates the throw statement, which consists of reserved word throw followed by an instance of Throwable or a subclass. (You will typically instantiate an Exception subclass.) This statement throws the instance to the virtual machine, which then searches for a suitable handler to handle the exception.

The first use of the throw statement is to throw a java.lang.NullPointerException instance when a null reference is passed as the source or destination filename. This unchecked exception is commonly thrown to indicate that a contract has been violated via a passed null reference. For example, you cannot pass null filenames to convert().

The second use of the throw statement is to throw a media.InvalidMediaFormatException instance when the expected media format doesn’t match the existing format. In the contrived example, the exception is thrown because the expected format is RM and the existing format is WAVE.

Unlike InvalidMediaFormatException, NullPointerException is not listed in convert()’s throws clause because NullPointerException instances are unchecked. They can occur so frequently that it is too big a burden to force the developer to properly handle these exceptions. Instead, the developer should write code that minimizes their occurrences.

Although not thrown from convert(), IOException is listed in this method’s throws clause in preparation for refactoring this method to perform the conversion with the help of file-handling code.
NullPointerException is one kind of exception that is thrown when an argument proves to be invalid. The java.lang.IllegalArgumentException class generalizes the illegal argument scenario to include other kinds of illegal arguments. For example, the following method throws an IllegalArgumentException instance when a numeric argument is negative:

```java
public static double sqrt(double x)
{
    if (x < 0)
        throw new IllegalArgumentException(x + " is negative");
    // Calculate the square root of x.
}
```

There are a few additional items to keep in mind when working with throws clauses and throw statements:

- You can append a throws clause to a constructor and throw an exception from the constructor when something goes wrong while the constructor is executing. The resulting object will not be created.

- When an exception is thrown out of an application’s main() method, the virtual machine terminates the application and calls the exception’s printStackTrace() method to print, to the console, the sequence of nested method calls that was awaiting completion when the exception was thrown.

- If a superclass method declares a throws clause, the overriding subclass method doesn’t have to declare a throws clause. However, if the subclass method does declare a throws clause, the clause must not include the names of checked exception classes that are not also included in the superclass method’s throws clause, unless they are the names of exception subclasses. For example, given superclass method void foo() throws IOException {}, the overriding subclass method could be declared as void foo() {}, void foo() throws IOException {}, or void foo() throws FileNotFoundException {}; the java.io.FileNotFoundException class subclasses IOException.

- A checked exception class name doesn’t need to appear in a throws clause when the name of its superclass appears.

- The compiler reports an error when a method throws a checked exception and doesn’t also handle the exception or list the exception in its throws clause.

- If at all possible, don’t include the names of unchecked exception classes in a throws clause. These names are not required because such exceptions should never occur. Furthermore, they only clutter source code and possibly confuse someone who is trying to understand that code.

- You can declare a checked exception class name in a method’s throws clause without throwing an instance of this class from the method. (Perhaps the method has yet to be fully coded.) However, Java requires that you provide code to handle this exception, even though it is not thrown.
Handling Exceptions

A method indicates its intention to handle one or more exceptions by specifying a try statement that includes one or more appropriate catch blocks. The try statement consists of reserved word try followed by a brace-delimited body. You place code that throws exceptions into this block.

A catch block consists of reserved word catch, followed by a round bracket-delimited single-parameter list that specifies an exception class name, followed by a brace-delimited body. You place code that handles exceptions whose types match the type of the catch block’s parameter list’s exception class parameter in this block.

A catch block is specified immediately after a try block. When an exception is thrown, the virtual machine will search for a handler. It first examines the catch block to see whether its parameter type matches or is the superclass type of the exception that has been thrown.

If the catch block is found, its body executes and the exception is handled. Otherwise, the virtual machine proceeds up the method-call stack, looking for the first method whose try statement contains an appropriate catch block. This process continues unless a catch block is found or execution leaves the main() method.

The following example illustrates try and catch:

```
try
{
    int x = 1 / 0;
}
catch (ArithmeticException ae)
{
    System.out.println("attempt to divide by zero");
}
```

When execution enters the try block, an attempt is made to divide integer 1 by integer 0. The virtual machine responds by instantiating ArithmeticException and throwing this exception. It then detects the catch block, which is capable of handling thrown ArithmeticException objects, and transfers execution to this block, which invokes System.out.println() to output a suitable message; the exception is handled.

Because ArithmeticException is an example of an unchecked exception type, and because unchecked exceptions represent coding mistakes that must be fixed, you typically don’t catch them, as demonstrated previously. Instead, you would fix the problem that led to the thrown exception.

**Tip** You might want to name your catch block parameters using the abbreviated style shown in the preceding section. Not only does this convention result in more meaningful exception-oriented parameter names (ae indicates that an ArithmeticException has been thrown), it can help reduce compiler errors. For example, it is common practice to name a catch block’s parameter e, for convenience. (Why type a long name?) However, the compiler will report an error when a previously declared local variable or parameter also uses e as its name; multiple same-named local variables and parameters cannot exist in the same scope.
Handling Multiple Exception Types

You can specify multiple catch blocks after a try block. For example, Listing 5-27's `convert()` method specifies a throws clause indicating that `convert()` can throw `InvalidMediaFormatException`, which is currently thrown, and `IOException`, which will be thrown when `convert()` is refactored. This refactoring will result in `convert()` throwing `IOException` when it cannot read from the source file or write to the destination file and throwing `FileNotFoundException` (a subclass of `IOException`) when it cannot open the source file or create the destination file. All these exceptions must be handled, as demonstrated in Listing 5-28.

Listing 5-28. Handling Different Kinds of Exceptions

```java
import java.io.FileNotFoundException;
import java.io.IOException;
import media.InvalidMediaFormatException;
import media.Media;

public class Converter {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Converter srcfile dstfile");
            return;
        }
        try {
            Media.convert(args[0], args[1]);
        } catch (InvalidMediaFormatException imfe) {
            System.out.println("Unable to convert " + args[0] + " to " + args[1]);
            System.out.println("Expecting " + args[0] + " to conform to " +
                                imfe.getExpectedFormat() + " format.");
            System.out.println("However, " + args[0] + " conformed to " +
                                imfe.getExistingFormat() + " format.");
        } catch (FileNotFoundException fnfe) {
        }
        catch (IOException ioe) {
        }
    }
}
```

The call to Media’s `convert()` method in Listing 5-28 is placed in a try block because this method is capable of throwing an instance of the checked `InvalidMediaFormatException`, `IOException`, or `FileNotFoundException` class; checked exceptions must be handled or be declared to be thrown via a throws clause that is appended to the method.
The catch (InvalidMediaFormatException imfe) block's statements are designed to provide a descriptive error message to the user. A more sophisticated application would localize these names so that the user could read the message in the user's language. The developer-oriented detail message is not output because it is not necessary in this trivial application.

**Note** A developer-oriented detail message is typically not localized. Instead, it is expressed in the developer's language. Users should never see detail messages.

Although not thrown, a catch block for IOException is required because this checked exception type appears in convert()'s throws clause. Because the catch (IOException ioe) block can also handle thrown FileNotFoundException instances (because FileNotFoundException subclasses IOException), the catch (FileNotFoundException fnfe) block isn't necessary at this point but is present to separate out the handling of a situation where a file cannot be opened for reading or created for writing (which will be addressed once convert() is refactored to include file code).

Assuming that the current directory contains Listing 5-28 and a media subdirectory containing InvalidMediaFormatException.java and Media.java, compile this listing (javac Converter.java), which also compiles media's source files, and run the application, as in java Converter A B. Converter responds by presenting the following output:

Unable to convert A to B
Expecting A to conform to RM format.
However, A conformed to WAVE format.

Listing 5-28's empty FileNotFoundException and IOException catch blocks illustrate the often-seen problem of leaving catch blocks empty because they are inconvenient to code. Unless you have a good reason, don't create an empty catch block. It swallows exceptions and you don't know that the exceptions were thrown. (For brevity, I don't always code catch blocks in this book's examples.)

**Caution** The compiler reports an error when you specify two or more catch blocks with the same parameter type after a try body. Example: try {} catch (IOException ioe1) {} catch (IOException ioe2) {}. You must merge these catch blocks into one block.

Although you can write catch blocks in any order, the compiler restricts this order when one catch block's parameter is a supertype of another catch block's parameter. The subtype parameter catch block must precede the supertype parameter catch block; otherwise, the subtype parameter catch block will never be executed.

For example, the FileNotFoundException catch block must precede the IOException catch block. If the compiler allowed the IOException catch block to be specified first, the FileNotFoundException catch block would never execute because a FileNotFoundException instance is also an instance of its IOException superclass.
Rethrowing Exceptions

While discussing the Throwable class, I discussed wrapping lower-level exceptions in higher-level exceptions. This activity will typically take place in a catch block and is illustrated in the following example:

```java
catch (IOException ioe)
{
    throw new ReportCreationException(ioe);
}
```

This example assumes that a helper method has just thrown a generic IOException instance as the result of trying to create a report. The public method's contract states that ReportCreationException is thrown in this case. To satisfy the contract, the latter exception is thrown. To satisfy the developer who is responsible for debugging a faulty application, the IOException instance is wrapped inside the ReportCreationException instance that is thrown to the public method's caller.

Sometimes, a catch block might not be able to fully handle an exception. Perhaps it needs access to information provided by some ancestor method in the method-call stack. However, the catch block might be able to partly handle the exception. In this case, it should partly handle the exception, and then rethrow the exception so that a handler in an ancestor method can finish handling it. Another possibility is to log the exception (for later analysis), which is demonstrated in the following example:

```java
catch (FileNotFoundException fnfe)
{
    logger.log(fnfe);
    throw fnfe; // Rethrow the exception here.
}
```

Performing Cleanup

In some situations, you might want to execute cleanup code before execution leaves a method following a thrown exception. For example, you might want to close a file that was opened, but could not be written, possibly because of insufficient disk space. Java provides the finally block for this situation.

The finally block consists of reserved word finally followed by a body, which provides the cleanup code. A finally block follows either a catch block or a try block. In the former case, the exception may be handled (and possibly rethrown) before finally executes. In the latter case, the exception is handled (and possibly rethrown) after finally executes.

Listing 5-29 demonstrates the first scenario in the context of a simulated file-copying application's main() method.

Listing 5-29. Cleaning Up by Closing Files After Handling a Thrown Exception

```java
import java.io.IOException;

public class Copy
{
    public static void main(String[] args)
```
{  
  if (args.length != 2)  
  {  
    System.err.println("usage: java Copy srcFile dstFile");  
    return;  
  }  
  
  int fileHandleSrc = 0;  
  int fileHandleDst = 1;  
  try  
  {  
    fileHandleSrc = open(args[0]);  
    fileHandleDst = create(args[1]);  
    copy(fileHandleSrc, fileHandleDst);  
  }  
  catch (IOException ioe)  
  {  
    System.err.println("I/O error: " + ioe.getMessage());  
    return;  
  }  
  finally  
  {  
    close(fileHandleSrc);  
    close(fileHandleDst);  
  }  
}  

static int open(String filename)  
{  
  return 1;  // Assume that filename is mapped to integer.  
}  

static int create(String filename)  
{  
  return 2;  // Assume that filename is mapped to integer.  
}  

static void close(int fileHandle)  
{  
  System.out.println("closing file: " + fileHandle);  
}  

static void copy(int fileHandleSrc, int fileHandleDst) throws IOException  
{  
  System.out.println("copying file " + fileHandleSrc + " to file " +  
      fileHandleDst);  
  if (Math.random() < 0.5)  
    throw new IOException("unable to copy file");  
}
Listing 5-29 presents a Copy application class that simulates the copying of bytes from a source file to a destination file. The try block invokes the open() method to open the source file and the create() method to create the destination file. Each method returns an integer-based file handle that uniquely identifies the file.

Next, this block calls the copy() method to perform the copy. After outputting a suitable message, copy() invokes the Math class’s random() method (officially discussed in Chapter 7) to return a random number between 0 and 1. When this method returns a value less than 0.5, which simulates a problem (perhaps the disk is full), the IOException class is instantiated and this instance is thrown.

The virtual machine locates the catch block that follows the try block and causes its handler to execute, which outputs a message. Then, the code in the finally block that follows the catch block is allowed to execute. Its purpose is to close both files by invoking the close() method on the passed file handle.

Compile this source code (javac Copy.java) and run the application with two arbitrary arguments (java Copy x.txt x.bak). You should observe the following output when there is no problem:

copying file 1 to file 2
closing file: 1
closing file: 2

When something goes wrong, you should observe the following output:

copying file 1 to file 2
I/O error: unable to copy file
closing file: 1
closing file: 2

Whether or not an I/O error occurs, notice that the finally block is the final code to execute. The finally block executes even though the catch block ends with a return statement.

This example illustrates finally block execution after a thrown exception is handled. However, you might want to perform cleanup before the exception is handled. Listing 5-30 presents a variation of the Copy application that demonstrates this alternative.

Listing 5-30. Cleaning Up by Closing Files Before Handling a Thrown Exception

import java.io.IOException;

public class Copy
{
    public static void main(String[] args) throws IOException
    {
        if (args.length != 2)
        {
            System.err.println("usage: java Copy srcFile dstFile");
            return;
        }
Listing 5-30 is nearly identical to Listing 5-29. The only difference is the throws clause appended to the main() method header and the removal of the catch block. When IOException is thrown, the finally block executes before execution leaves the main() method. This time, Java's default exception handler executes printStackTrace() and you observe output similar to the following:

copying file 1 to file 2
closing file: 1
closing file: 2
Exception in thread "main" java.io.IOException: unable to copy file
  at Copy.copy(Copy.java:48)
  at Copy.main(Copy.java:19)
EXERCISES

The following exercises are designed to test your understanding of Chapter 5’s content.

1. What is a nested class?
2. Identify the four kinds of nested classes.
3. Which nested classes are also known as inner classes?
4. True or false: A static member class has an enclosing instance.
5. How do you instantiate a nonstatic member class from beyond its enclosing class?
6. When is it necessary to declare local variables and parameters final?
7. True or false: An interface can be declared within a class or within another interface.
8. Define package.
9. How do you ensure that package names are unique?
10. What is a package statement?
11. True or false: You can specify multiple package statements in a source file.
12. What is an import statement?
13. How do you indicate that you want to import multiple types via a single import statement?
14. During a runtime search, what happens when the virtual machine cannot find a classfile?
15. How do you specify the user classpath to the virtual machine?
17. Why are constant interfaces used?
18. Why are constant interfaces bad?
19. What is a static import statement?
20. How do you specify a static import statement?
21. What is an exception?
22. In what ways are objects superior to error codes for representing exceptions?
23. What is a throwable?
24. What does the getCause() method return?
25. What is the difference between Exception and Error?
26. What is a checked exception?
27. What is a runtime exception?
28. Under what circumstance would you introduce your own exception class?
29. True or false: You use a throw statement to identify exceptions that are thrown from a method by appending this statement to a method’s header.
30. What is the purpose of a try statement, and what is the purpose of a catch block?
31. What is the purpose of a finally block?
32. A 2D graphics package supports two-dimensional drawing and transformations (rotation, scaling, translation, etc.). These transformations require a 3-by-3 matrix (a table). Declare a G2D class that encloses a private Matrix nonstatic member class. Instantiate Matrix within G2D’s noargument constructor, and initialize the Matrix instance to the identity matrix (a matrix where all entries are 0 except for those on the upper-left to lower-right diagonal, which are 1).

33. Extend the logging package to support a null device in which messages are thrown away.

34. Modify the logging package so that Logger’s connect() method throws CannotConnectException when it cannot connect to its logging destination, and the other two methods each throw NotConnectedException when connect() was not called or when it threw CannotConnectException.

35. Modify TestLogger to respond appropriately to thrown CannotConnectException and NotConnectedException objects.

Summary

Classes that are declared outside of any class are known as top-level classes. Java also supports nested classes, which are classes that are declared as members of other classes or scopes.

There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. The latter three categories are known as inner classes.

Java supports the partitioning of top-level types into multiple namespaces, to better organize these types and to also prevent name conflicts. Java uses packages to accomplish these tasks.

The package statement identifies the package in which a source file’s types are located. The import statement imports types from a package by telling the compiler where to look for unqualified type names during compilation.

An exception is a divergence from an application’s normal behavior. Although it can be represented by an error code or object, Java uses objects because error codes are meaningless and cannot contain information about what led to the exception.

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in Throwable. Moving down the throwable hierarchy, you encounter the Exception and Error classes, which represent nonerror exceptions and errors.

Exception and its subclasses, except for RuntimeException (and its subclasses), describe checked exceptions. They are checked because you must check the code to ensure that an exception is handled where thrown or identified as being handled elsewhere.

RuntimeException and its subclasses describe unchecked exceptions. You don’t have to handle these exceptions because they represent coding mistakes (fix the mistakes). Although the names of their classes can appear in throws clauses, doing so adds clutter.

The throw statement throws an exception to the virtual machine, which searches for an appropriate handler. When the exception is checked, its name must appear in the method’s throws clause, unless the name of the exception’s superclass is listed in this clause.

A method handles one or more exceptions by specifying a try statement and appropriate catch blocks. A finally block can be included to execute cleanup code whether an exception is thrown or not, and before a thrown exception leaves the method.

Chapter 6 continues to explore the Java language by focusing on assertions, annotations, generics, and enums.
Mastering Advanced Language Features, Part 2

In Chapters 2 through 4, I laid a foundation for learning the Java language, and in Chapter 5, I built onto this foundation by introducing some of Java’s more advanced language features. In this chapter, I will continue to cover advanced language features by focusing on those features related to assertions, annotations, generics, and enums.

Mastering Assertions

Writing source code is not an easy task. All too often, bugs are introduced into the code. When a bug is not discovered before compiling the source code, it makes it into runtime code, which will probably fail unexpectedly (or show no sign of failure but give wrong output). At this point, the cause of failure can be very difficult to determine.

Developers often make assumptions about application correctness, and some developers think that specifying comments that state their beliefs about what they think is true at the comment locations is sufficient for determining correctness. However, comments are useless for preventing bugs because the compiler ignores them.

Many languages address this problem by providing a language feature called assertions that lets the developer codify assumptions about application correctness. When the application runs, and if an assertion fails, the application terminates with a message that helps the developer diagnose the failure’s cause. (You might think of assertions as comments that the compiler understands.)
In his “Assert Statements Shine Light Into Dark Corners” blog post (www.drdobbs.com/cpp/assert-statements-shine-light-into-dark/240012746), computer scientist Andrew Koenig mentions that assertions are used to detect invariant failures, where an invariant is something in your code that should not change. For example, you might want to verify the expectation that a list of data items is sorted (an invariant) before attempting to search that list via the Binary Search algorithm, which requires that the list be sorted. You would use an assertion to learn whether the invariant holds or not.

In this section, I will introduce you to Java’s assertions language feature. After defining this term, I will show you how to declare assertions and then look at how to use and avoid assertions. Finally, you will learn how to selectively enable and disable assertions via the javac compiler tool’s command-line arguments.

**Declaring Assertions**

An assertion is a statement that lets you express an assumption of program correctness via a Boolean expression. If this expression evaluates to true, execution continues with the next statement. Otherwise, an error that identifies the cause of failure is thrown.

There are two forms of the assertion statement, with each form beginning with reserved word assert.

```java
assert expression1;
assert expression1 : expression2;
```

In both forms of this statement, `expression1` is the Boolean expression. In the second form, `expression2` is any expression that returns a value. It cannot be a call to a method whose return type is `void`.

When `expression1` evaluates to false, this statement instantiates class `java.lang.AssertionError`. The first statement form calls this class’s noargument constructor, which doesn’t associate a message identifying failure details with the `AssertionError` instance. The second form calls an `AssertionError` constructor whose type matches the type of `expression2`’s value. This value is passed to the constructor and its string representation is used as the error’s detail message.

When the error is thrown, the name of the source file and the number of the line from where the error was thrown are output to the console as part of the thrown error’s stack trace. In many situations, this information is sufficient for identifying what led to the failure, and the first form of the assertion statement should be used.

Listing 6-1 demonstrates the first form of the assertion statement.

**Listing 6-1. Throwing an Assertion Error without a Detail Message**

```java
public class AssertionDemo
{
    public static void main(String[] args)
    {
        int x = 1;
        assert x == 0;
    }
}
```
When assertions are enabled (I will discuss this task later), running the previous application results in the following output:

```
Exception in thread "main" java.lang.AssertionError
    at AssertionDemo.main(AssertionDemo.java:6)
```

In other situations, more information is needed to help diagnose the cause of failure. For example, suppose `expression1` compares variables `x` and `y`, and throws an error when `x`'s value exceeds `y`'s value. Because this should never happen, you would probably use the second statement form to output these values so you could diagnose the problem.

Listing 6-2 demonstrates the second form of the assertion statement.

**Listing 6-2. Throwing an Assertion Error with a Detail Message**

```java
public class AssertionDemo
{
    public static void main(String[] args)
    {
        int x = 1;
        assert x == 0: x;
    }
}
```

Once again, it is assumed that assertions are enabled. Running the previous application results in the following output:

```
Exception in thread "main" java.lang.AssertionError: 1
    at AssertionDemo.main(AssertionDemo.java:6)
```

The value in `x` is appended to the end of the first output line, which is somewhat cryptic. To make this output more meaningful, you might want to specify an expression that also includes the variable's name: `assert x == 0: "x = " + x;`, for example.

**Using Assertions**

There are many situations where assertions should be used. These situations organize into internal invariant, control-flow invariant, and design-by-contract categories. An *invariant* is something in your code that should not change.

**Internal Invariants**

An *internal invariant* is expression-oriented behavior that is not expected to change. For example, Listing 6-3 introduces an internal invariant by way of chained if-else statements that output the state of water based on its temperature.
Listing 6-3. Discovering That an Internal Invariant Can Vary

```java
public class IIDemo
{
    public static void main(String[] args)
    {
        double temperature = 50.0; // Celsius
        if (temperature < 0.0)
            System.out.println("water has solidified");
        else
            if (temperature >= 100.0)
                System.out.println("water is boiling into a gas");
            else
            {
                // temperature > 0.0 and temperature < 100.0
                assert(temperature > 0.0 && temperature < 100.0): temperature;
                System.out.println("water is remaining in its liquid state");
            }
    }
}
```

A developer might specify only a comment stating an assumption as to what expression causes the final else to be reached. Because the comment might not be enough to detect the lurking < 0.0 expression bug (water is also solid at zero degrees), an assertion statement is necessary.

Another example of an internal invariant concerns a switch statement with no default case. The default case is avoided because the developer believes that all paths have been covered. However, this is not always true, as Listing 6-4 demonstrates.

Listing 6-4. Another Buggy Internal Invariant

```java
public class IIDemo
{
    final static int NORTH = 0;
    final static int SOUTH = 1;
    final static int EAST = 2;
    final static int WEST = 3;

    public static void main(String[] args)
    {
        int direction = (int) (Math.random() * 5);
        switch (direction)
        {
            case NORTH: System.out.println("travelling north"); break;
            case SOUTH: System.out.println("travelling south"); break;
            case EAST : System.out.println("travelling east"); break;
            case WEST : System.out.println("travelling west"); break;
            default   : assert false;
        }
    }
}
```
Listing 6-4 assumes that the expression tested by switch will only evaluate to one of four integer constants. However, \((\text{int}) (\text{Math.random() \ast 5})\) can also return 4, causing the default case to execute \texttt{assert false}, which always throws \texttt{AssertionError}. (You might have to run this application a few times to see the assertion error, but first you need to learn how to enable assertions, which I will discuss later in this chapter.)

**Tip** When assertions are disabled, \texttt{assert false;} doesn't execute and the bug goes undetected. To always detect this bug, replace \texttt{assert false;} with \texttt{throw new AssertionError(direction);}.

### Control-Flow Invariants

An *control-flow invariant* is a flow of control that is not expected to change. For example, Listing 6-4 uses an assertion to test an assumption that switch’s default case will not execute. Listing 6-5, which fixes Listing 6-4’s bug, provides another example.

**Listing 6-5. A Buggy Control-Flow Invariant**

```java
public class CFDemo
{
    final static int NORTH = 0;
    final static int SOUTH = 1;
    final static int EAST = 2;
    final static int WEST = 3;

    public static void main(String[] args)
    {
        int direction = (int) (Math.random() \ast 4);
        switch (direction)
        {
            case NORTH: System.out.println("travelling north"); break;
            case SOUTH: System.out.println("travelling south"); break;
            case EAST : System.out.println("travelling east"); break;
            case WEST : System.out.println("travelling west");
            default   : assert false;
        }
    }
}
```

Because the original bug has been fixed, the default case should never be reached. However, the omission of a break statement that terminates \texttt{case WEST} causes execution to reach the default case. This control-flow invariant has been broken. (Again, you might have to run this application a few times to see the assertion error, but first you need to learn how to enable assertions, which I will discuss later in this chapter.)
Caution  Be careful when using an assertion statement to detect code that should never be executed. If the assertion statement cannot be reached according to the rules set forth in *The Java Language Specification, Third Edition*, by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha (Addison-Wesley, 2005; ISBN: 0321246780; also available at [http://docs.oracle.com/javase/specs/](http://docs.oracle.com/javase/specs/)), the compiler will report an error. For example, for (;;) { assert false; } causes the compiler to report an error because the infinite for loop prevents the assertion statement from executing.

Design-by-Contract


Preconditions

A *precondition* is something that must be true when a method is called. Assertion statements are often used to satisfy a helper method’s preconditions by checking that its arguments are legal. Listing 6-6 provides an example.

**Listing 6-6. Verifying a Precondition**

```java
public class Lotto649 {
    public static void main(String[] args) {
        // Lotto 649 requires that six unique numbers be chosen.
        int[] selectedNumbers = new int[6];
        // Assign a unique random number from 1 to 49 (inclusive) to each slot
        // in the selectedNumbers array.
        for (int slot = 0; slot < selectedNumbers.length; slot++) {
            int num;
            // Obtain a random number from 1 to 49. That number becomes the
            // selected number if it has not previously been chosen.
            try_again:
            do {
                num = rnd(49) + 1;
                for (int i = 0; i < slot; i++)
                    if (selectedNumbers[i] == num)
                        continue try_again;
                break;
            } while (true);
            // Assign selected number to appropriate slot.
            selectedNumbers[slot] = num;
        }
    }
```


// Sort all selected numbers into ascending order and then print these
// numbers.
    sort(selectedNumbers);
    for (int i = 0; i < selectedNumbers.length; i++)
        System.out.print(selectedNumbers[i] + " ");

}

static int rnd(int limit)
{
    // This method returns a random number (actually, a pseudorandom number)
    // ranging from 0 through limit - 1 (inclusive).
    assert limit > 1: "limit = " + limit;
    return (int) (Math.random() * limit);
}

static void sort(int[] x)
{
    // This method sorts the integers in the passed array into ascending
    // order.
    for (int pass = 0; pass < x.length - 1; pass++)
        for (int i = x.length - 1; i > pass; i--)
            if (x[i] < x[pass])
                {
                    int temp = x[i];
                    x[i] = x[pass];
                    x[pass] = temp;
                }
}

Listing 6-6’s application simulates Lotto 6/49, one of Canada’s national lottery games. The rnd() helper method returns a randomly chosen integer between 0 and limit - 1. An assertion statement verifies the precondition that limit’s value must be 2 or higher.

Note The sort() helper method sorts (orders) the selectedNumbers array’s integers into ascending order by implementing an algorithm (a recipe for accomplishing some task) called Bubble Sort.

Bubble Sort works by making multiple passes over the array. During each pass, various comparisons and swaps ensure that the next smallest element value “bubbles” toward the top of the array, which would be the element at index 0.

Bubble Sort is not efficient, but is more than adequate for sorting a six-element array. Although I could have used one of the efficient sort() methods located in the java.util package’s Arrays class (for example, Arrays.sort(selectedNumbers)); accomplishes the same objective as Listing 6-6’s sort(selectedNumbers); method call, but does so more efficiently), I chose to use Bubble Sort because I prefer to wait until Chapter 9 before getting into the Arrays class.
Postconditions

A postcondition is something that must be true after a method successfully completes. Assertion statements are often used to satisfy a helper method’s postconditions by checking that its result is legal. Listing 6-7 provides an example.

Listing 6-7. Verifying a Postcondition in Addition to Preconditions

```java
public class MergeArrays
{
    public static void main(String[] args)
    {
        int[] x = { 1, 2, 3, 4, 5 };  
        int[] y = { 1, 2, 7, 9 };  
        int[] result = merge(x, y);  
        for (int i = 0; i < result.length; i++)  
            System.out.println(result[i]);  
    }

    static int[] merge(int[] a, int[] b)
    {
        if (a == null)
            throw new NullPointerException("a is null");
        if (b == null)
            throw new NullPointerException("b is null");
        int[] result = new int[a.length + b.length];
        // Precondition
        assert result.length == a.length + b.length: "length mismatch";
        for (int i = 0; i < a.length; i++)
            result[i] = a[i];
        for (int i = 0; i < b.length; i++)
            result[a.length + i - 1] = b[i];
        // Postcondition
        assert containsAll(result, a, b): "value missing from array";
        return result;
    }

    static boolean containsAll(int[] result, int[] a, int[] b)
    {
        for (int i = 0; i < a.length; i++)
            if (!contains(result, a[i]))
                return false;
        for (int i = 0; i < b.length; i++)
            if (!contains(result, b[i]))
                return false;
        return true;
    }
```
static boolean contains(int[] a, int val) {
    for (int i = 0; i < a.length; i++)
        if (a[i] == val)
            return true;
    return false;
}

Listing 6-7 uses an assertion statement to verify the postcondition that all of the values in the two arrays being merged are present in the merged array. The postcondition is not satisfied, however, because this listing contains a bug.

Listing 6-7 also shows preconditions and postconditions being used together. The solitary precondition verifies that the merged array length equals the lengths of the arrays being merged prior to the merge logic.

Class Invariants

A class invariant is a kind of internal invariant that applies to every instance of a class at all times, except when an instance is transitioning from one consistent state to another.

For example, suppose instances of a class contain arrays whose values are sorted in ascending order. You might want to include an isSorted() method in the class that returns true when the array is still sorted, and verify that each constructor and method that modifies the array specifies assert isSorted(); prior to exit, to satisfy the assumption that the array is still sorted when the constructor or method exits.

Avoiding Assertions

Although there are many situations where assertions should be used, there also are situations where they should be avoided. For example, you should not use assertions to check the arguments that are passed to public methods for the following reasons:

- Checking a public method’s arguments is part of the contract that exists between the method and its caller. If you use assertions to check these arguments, and if assertions are disabled, this contract is violated because the arguments will not be checked.

- Assertions also prevent appropriate exceptions from being thrown. For example, when an illegal argument is passed to a public method, it is common to throw java.lang.IllegalArgumentException or java.lang.NullPointerException. However, AssertionError is thrown instead.

You should also avoid using assertions to perform work required by the application to function correctly. This work is often performed as a side effect of the assertion’s Boolean expression. When assertions are disabled, the work is not performed.

For example, suppose you have a list of Employee objects and a few null references that are also stored in this list, and you want to remove all of the null references. It would not be correct to remove these references via the following assertion statement:

assert employees.removeAll(null);
Although the assertion statement will not throw AssertionError because there is at least one null reference in the employees list, the application that depends upon this statement executing will fail when assertions are disabled.

Instead of depending on the former code to remove the null references, you would be better off using code similar to the following:

```java
boolean allNullsRemoved = employees.removeAll(null);
assert allNullsRemoved;
```

This time, all null references are removed regardless of whether assertions are enabled or disabled and you can still specify an assertion to verify that nulls were removed.

### Enabling and Disabling Assertions

The compiler records assertions in the classfile. However, assertions are disabled at runtime because they can affect performance. An assertion might call a method that takes awhile to complete, and this would impact the running application's performance.

You must enable the classfile's assertions before you can test assumptions about the behaviors of your classes. Accomplish this task by specifying the `-enableassertions` or `-ea` command-line option when running the `java` application launcher tool.

The `-enableassertions` and `-ea` command-line options let you enable assertions at various granularities based upon one of the following arguments (except for the noargument scenario, you must use a colon to separate the option from its argument):

- **No argument**: Assertions are enabled in all classes except system classes.
- **PackageName...**: Assertions are enabled in the specified package and its subpackages by specifying the package name followed by `...`.
- **...**: Assertions are enabled in the unnamed package, which happens to be whatever directory is current.
- **ClassName**: Assertions are enabled in the named class by specifying the class name.

For example, you can enable all assertions except system assertions when running the MergeArrays application via `java -ea MergeArrays`. Also, you could enable any assertions that you might add to Chapter 5's logging package by specifying `java -ea:logging TestLogger`.

Assertions can be disabled, and also at various granularities, by specifying either of the `-disableassertions` or `-da` command-line options. These options take the same arguments as `-enableassertions` and `-ea`. For example, `java -ea -da:loneclass mainclass` enables all assertions except for those in `loneclass` (Think of `loneclass` and `mainclass` as placeholders for the actual classes that you specify."

The previous options apply to all classloaders (discussed in Chapter 16). Except when taking no arguments, they also apply to system classes. This exception simplifies the enabling of assertion statements in all classes except for system classes, which is often desirable.
To enable system assertions, specify either -enablesystemassertions or -esa, for example, java -esa -ea:logging TestLogger. Specify either -disablesystemassertions or -dsa to disable system assertions.

Mastering Annotations

While developing a Java application, you might want to annotate (associate metadata, which is data that describes other data, with) various application elements. For example, you might want to identify methods that are not fully implemented so that you will not forget to implement them. Java's annotations language feature lets you accomplish this task.

In this section I will introduce you to annotations. After defining this term and presenting three kinds of compiler-supported annotations as examples, I will show you how to declare your own annotation types and use these types to annotate source code. Finally, you will discover how to process your own annotations to accomplish useful tasks.

Note Java has always supported ad hoc annotation mechanisms. For example, the java.lang.Cloneable interface identifies classes whose instances can be shallowly cloned via java.lang.Object's clone() method; the transient reserved word marks fields that are to be ignored during serialization; and the @deprecated Javadoc tag documents methods that are no longer supported. In contrast, the annotations feature is a standard for annotating code.

Discovering Annotations

An annotation is an instance of an annotation type and associates metadata with an application element. It is expressed in source code by prefixing the type name with the @ symbol. For example, @Readonly is an annotation and Readonly is its type.

Note You can use annotations to associate metadata with constructors, fields, local variables, methods, packages, parameters, and types (annotation, class, enum, and interface).

The compiler supports the Override, Deprecated, and SuppressWarnings annotation types. These types are located in the java.lang package.

@Override annotations are useful for expressing that a subclass method overrides a method in the superclass and doesn't overload that method instead. The following example reveals this annotation being used to prefix the overriding method:

```java
@Override
public void draw(int color)
{
    // drawing code
}
```
@Deprecated annotations are useful for indicating that the marked application element is deprecated (phased out) and should no longer be used. The compiler warns you when a deprecated application element is accessed by nondeprecated code.

In contrast, the @deprecated javadoc tag and associated text warns you against using the deprecated item, and tells you what to use instead. The following example demonstrates that @Deprecated and @deprecated can be used together:

```java
/**
 * Allocates a <code>Date</code> object and initializes it so that
 * it represents midnight, local time, at the beginning of the day
 * specified by the <code>year</code>, <code>month</code>, and
 * <code>date</code> arguments.
 * *
 * @param year the year minus 1900.
 * @param month the month between 0-11.
 * @param date the day of the month between 1-31.
 * @see java.util.Calendar
 * @deprecated As of JDK version 1.1,
 * replaced by <code>Calendar.set(year + 1900, month, date)</code>
 * or <code>GregorianCalendar(year + 1900, month, date)</code>.
 */
@Deprecated
public Date(int year, int month, int date)
{
    this(year, month, date, 0, 0, 0);
}
```

This example excerpts one of the constructors in Java's Date class (located in the java.util package). Its Javadoc comment reveals that Date(int year, int month, int date) has been deprecated in favor of using the set() method in the Calendar class (also located in the java.util package). (I explore Date and Calendar in Chapter 16.)

The compiler suppresses warnings when a compilation unit (typically a class or interface) refers to a deprecated class, method, or field. This feature lets you modify legacy APIs without generating deprecation warnings and is demonstrated in Listing 6-8.

Listing 6-8. Referencing a Deprecated Field from the Same Class Declaration

```java
public class Employee
{
    /**
     * Employee's name
     * @deprecated New version uses firstName and lastName fields.
     */
    @Deprecated
    String name;
    String firstName;
    String lastName;
}
public static void main(String[] args) {
    Employee emp = new Employee();
    emp.name = "John Doe";
}

Listing 6-8 declares an Employee class with a name field that has been deprecated. Although Employee's main() method refers to name, the compiler will suppress a deprecation warning because the deprecation and reference occur in the same class.

Suppose you refactor this listing by introducing a new UseEmployee class and moving Employee's main() method to this class. Listing 6-9 presents the resulting class structure.

Listing 6-9. Referencing a Deprecated Field from Another Class Declaration

class Employee {
    /**
     * Employee's name
     * @deprecated New version uses firstName and lastName fields.
     */
    @Deprecated
    String name;
    String firstName;
    String lastName;
}

class UseEmployee {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.name = "John Doe";
    }
}

If you attempt to compile this source code via the javac compiler tool, you will discover the following messages:

Note: UseEmployee.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

You will need to specify -Xlint:deprecation as one of javac's command-line arguments (as in javac -Xlint:deprecation UseEmployee.java) to discover the deprecated item and the code that refers to this item:

Employee.java:18: warning: [deprecation] name in Employee has been deprecated
    emp.name = "John Doe";
    ^
1 warning
@SuppressWarnings annotations are useful for suppressing deprecation or unchecked warnings via a "deprecation" or an "unchecked" argument. (Unchecked warnings occur when mixing code that uses generics with pre-generics legacy code. I will discuss generics and unchecked warnings later in this chapter.)

For example, Listing 6-10 uses @SuppressWarnings with a "deprecation" argument to suppress the compiler’s deprecation warnings when code in the UseEmployee class’s main() method accesses the Employee class’s name field.

Listing 6-10. Suppressing the Previous Deprecation Warning

```java
public class UseEmployee {
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.name = "John Doe";
    }
}
```

Note As a matter of style, you should always specify @SuppressWarnings on the most deeply nested element where it is effective. For example, if you want to suppress a warning in a particular method, you should annotate that method rather than its class.

## Declaring Annotation Types and Annotating Source Code

Before you can annotate source code, you need annotation types that can be instantiated. Java supplies many annotation types in addition to Override, Deprecated, and SuppressWarnings. Java also lets you declare your own types.

You declare an annotation type by specifying the @ symbol, immediately followed by reserved word interface, followed by the type’s name, followed by a body. For example, Listing 6-11 uses @interface to declare an annotation type named Stub.

Listing 6-11. Declaring the Stub Annotation Type

```java
public @interface Stub {
}
```

Instances of annotation types that supply no data apart from a name—their bodies are empty—are known as marker annotations because they mark application elements for some purpose. As Listing 6-12 reveals, @Stub is used to mark empty methods (stubs).
Listing 6-12. Annotating a Stubbed-Out Method

public class Deck // Describes a deck of cards.
{
    @Stub
    public void shuffle()
    {
        // This method is empty and will presumably be filled in with appropriate
        // code at some later date.
    }
}

Listing 6-12’s Deck class declares an empty shuffle() method. This fact is indicated by instantiating Stub and prefixing shuffle()’s method header with the resulting @Stub annotation.

Note Although marker interfaces (introduced in Chapter 4) appear to have been replaced by marker annotations, this is not the case, because marker interfaces have advantages over marker annotations. One advantage is that a marker interface specifies a type that is implemented by a marked class, which lets you catch problems at compile time. For example, when a class doesn’t implement the Cloneable interface, its instances cannot be shallowly cloned via Object’s clone() method. If Cloneable had been implemented as a marker annotation, this problem would not be detected until runtime.

Although marker annotations are useful (@Override and @Deprecated are good examples), you will typically want to enhance an annotation type so that you can store metadata via its instances. You accomplish this task by adding elements to the type.

An element is a method header that appears in the annotation type’s body. It cannot have parameters or a throws clause, and its return type must be a primitive type (such as int), java.lang.String, java.lang.Class, an enum (discussed later in this chapter), an annotation type, or an array of the preceding types. However, it can have a default value.

Listing 6-13 adds three elements to Stub.

Listing 6-13. Adding Three Elements to the Stub Annotation Type

public @interface Stub
{
    int id(); // A semicolon must terminate an element declaration.
    String dueDate();
    String developer() default "unassigned";
}

The id() element specifies a 32-bit integer that identifies the stub. The dueDate() element specifies a String-based date that identifies when the method stub is to be implemented. Finally, developer() specifies the String-based name of the developer responsible for coding the method stub.
Unlike `id()` and `dueDate()`, `developer()` is declared with a default value, "unassigned". When you instantiate `Stub` and don’t assign a value to `developer()` in that instance, as is the case with Listing 6-14, this default value is assigned to `developer()`.

Listing 6-14. Initializing a Stub Instance’s Elements

```java
public class Deck {
    @Stub(id = 1,
          dueDate = "12/21/2012")
    public void shuffle()
    {
    }
}
```

Listing 6-14 reveals one `@Stub` annotation that initializes its `id()` element to 1 and its `dueDate()` element to "12/21/2012". Each element name doesn’t have a trailing `()`, and the comma-separated list of two element initializers appears between `( and )`.

Suppose you decide to replace `Stub`’s `id()`, `dueDate()`, and `developer()` elements with a single `String` `value()` element whose string specifies comma-separated ID, due date, and developer name values. Listing 6-15 shows you two ways to initialize `value`.

Listing 6-15. Initializing Each Stub Instance’s `value()` Element

```java
public class Deck {
    @Stub(value = "1,12/21/2012,unassigned")
    public void shuffle()
    {
    }

    @Stub("2,12/21/2012,unassigned")
    public Card[] deal(int ncards)
    {
        return null;
    }
}
```

Listing 6-15 reveals special treatment for the `value()` element. When it’s an annotation type’s only element, you can omit `value()`’s name and `=` from the initializer. I used this fact to specify `@SuppressWarnings("deprecation")` in Listing 6-10.

Using Meta-Annotations in Annotation Type Declarations

Each of the `Override`, `Deprecated`, and `SuppressWarnings` annotation types is itself annotated with `meta-annotations` (annotations that annotate annotation types). For example, Listing 6-16 shows you that the `SuppressWarnings` annotation type is annotated with two meta-annotations.

```java
import java.lang.annotation.
```
Listing 6-16. The annotated SuppressWarnings Type Declaration

```java
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

The Target annotation type, which is located in the java.lang.annotation package, identifies the kinds of application elements to which an annotation type applies. @Target indicates that @SuppressWarnings annotations can be used to annotate types, fields, methods, parameters, constructors, and local variables.

Each of `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, and `LOCAL_VARIABLE` is a member of the ElementType enum, which is also located in the java.lang.annotation package. (I discuss enums later in this chapter.)

The `{` and `}` characters surrounding the comma-separated list of values assigned to Target’s value() element signify an array—value()’s return type is `String[]`. Although these braces are necessary (unless the array consists of one item), value= could be omitted when initializing @Target because Target declares only a value() element.

The Retention annotation type, which is located in the java.lang.annotation package, identifies the retention (also known as lifetime) of an annotation type’s annotations. @Retention indicates that @SuppressWarnings annotations have a lifetime that is limited to source code—they don’t exist after compilation.

`SOURCE` is one of the members of the RetentionPolicy enum (located in the java.lang.annotation package). The other members are `CLASS` and `RUNTIME`. These three members specify the following retention policies:

- **CLASS**: The compiler records annotations in the classfile, but the virtual machine doesn’t retain them (to save memory space). This policy is the default.
- **RUNTIME**: The compiler records annotations in the classfile, and the virtual machine retains them so that they can be read via the Reflection API (see Chapter 8) at runtime.
- **SOURCE**: The compiler discards annotations after using them.

There are two problems with the Stub annotation type shown in Listings 6-11 and 6-13. First, the lack of an @Target meta-annotation means that you can annotate any application element @Stub. However, this annotation only makes sense when applied to methods and constructors. Check out Listing 6-17.

Listing 6-17. Annotating Undesirable Application Elements

```java
@Stub("1,12/21/2012,unassigned")
public class Deck
{
    @Stub("2,12/21/2012,unassigned")
    private Card[] cardsRemaining = new Card[52];
```
Listing 6-17 uses @Stub to annotate the Deck class, the cardsRemaining field, and the ncards parameter as well as annotating the constructor and the two methods. The first three application elements are inappropriate to annotate because they are not stubs.

You can fix this problem by prefixing the Stub annotation type declaration with @Target({ElementType.METHOD, ElementType.CONSTRUCTOR}) so that Stub only applies to methods and constructors. After doing this, the javac compiler tool will output the following error messages when you attempt to compile Listing 6-17:

```
Deck.java:1: error: annotation type not applicable to this kind of declaration
@Stub("1,12/21/2012,unassigned")
    ^
Deck.java:4: error: annotation type not applicable to this kind of declaration
    @Stub("2,12/21/2012,unassigned")
        ^
Deck.java:18: error: annotation type not applicable to this kind of declaration
    public Card[] deal(@Stub("5,12/21/2012,unassigned") int ncards)
        ^
3 errors
```

The second problem is that the default CLASS retention policy makes it impossible to process @Stub annotations at runtime. You can fix this problem by prefixing the Stub type declaration with @Retention(RetentionPolicy.RUNTIME).

Listing 6-18 presents the Stub annotation type with the desired @Target and @Retention meta-annotations.

```
Listing 6-18. A Revamped Stub Annotation Type

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Stub {
    String value();
}

Note  Java also provides Documented and Inherited meta-annotation types in the java.lang.annotation package. Instances of @Documented-annotated annotation types are to be documented by javadoc and similar tools, whereas instances of @Inherited-annotated annotation types are automatically inherited. According to Inherited's Java documentation, if “the user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, then the class's superclass will automatically be queried for the annotation type. This process will be repeated until an annotation for this type is found, or the top of the class hierarchy (Object) is reached. If no superclass has an annotation for this type, then the query will indicate that the class in question has no such annotation.”

Processing Annotations

It's not enough to declare an annotation type and use that type to annotate source code. Unless you do something specific with those annotations, they remain dormant. One way to accomplish something specific is to write an application that processes the annotations. Listing 6-19's StubFinder application does just that.

Listing 6-19. The StubFinder Application

import java.lang.reflect.Method;

public class StubFinder {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("usage: java StubFinder classfile");
            return;
        }
        Method[] methods = Class.forName(args[0]).getMethods();
        for (int i = 0; i < methods.length; i++)
            if (methods[i].isAnnotationPresent(Stub.class)) {
                Stub stub = methods[i].getAnnotation(Stub.class);
                String[] components = stub.value().split(",");
                System.out.println("Stub ID = " + components[0]);
                System.out.println("Stub Date = " + components[1]);
### Caution

There are two problems with throws Exception. First, it is often better to handle the exception and present a suitable error message than to “pass the buck” by throwing it out of main(). Second, Exception is generic; it hides the names of the kinds of exceptions that are thrown. However, I find it convenient to specify throws Exception in a throwaway utility.

---

After compiling StubFinder (javac StubFinder.java), Stub (javac Stub.java), and Listing 6-15’s Deck class (javac Deck.java), run StubFinder with Deck as its single command-line argument (java StubFinder Deck). You will observe the following output:

Stub ID = 1
Stub Date = 12/21/2012
Stub Developer = unassigned

Stub ID = 2
Stub Date = 12/21/2012
Stub Developer = unassigned
Mastering Generics

Java 5 introduced generics, language features for declaring and using type-agnostic classes and interfaces. While working with Java's Collections Framework (which I discuss in Chapter 9), these features help you avoid java.lang.ClassCastExceptions.

Note Although the main use for generics is the Collections Framework, the standard class library also contains generified (retrofitted to make use of generics) classes that have nothing to do with this framework: java.lang.Class, java.lang.ThreadLocal, and java.lang.ref.WeakReference are three examples.

In this section, I will introduce you to generics. You will first learn how generics promote type safety in the context of the Collections Framework classes, and then you will explore generics in the contexts of generic types and generic methods. Finally, you will learn about generics in the context of arrays.

Collections and the Need for Type Safety

Java’s Collections Framework makes it possible to store objects in various kinds of containers (known as collections) and later retrieve those objects. For example, you can store objects in a list, a set, or a map. You can then retrieve a single object, or iterate over the collection and retrieve all objects.

Before Java 5 overhauled the Collections Framework to take advantage of generics, there was no way to prevent a collection from containing objects of mixed types. The compiler didn’t check an object’s type to see if it was suitable before it was added to a collection, and this lack of static type checking led to ClassCastExceptions.

Listing 6-20 demonstrates how easy it is to generate a ClassCastException.

Listing 6-20. Lack of Type Safety Leading to a ClassCastException at Runtime

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

class Employee
{
    private String name;

    Employee(String name)
    {
        this.name = name;
    }
}
```
String getName()
{
    return name;
}

class TypeSafety
{
    public static void main(String[] args)
    {
        List employees = new ArrayList();
        employees.add(new Employee("John Doe"));
        employees.add(new Employee("Jane Smith"));
        employees.add("Jack Frost");
        Iterator iter = employees.iterator();
        while (iter.hasNext())
        {
            Employee emp = (Employee) iter.next();
            System.out.println(emp.getName());
        }
    }
}

Listing 6-20’s main() method first instantiates java.util.ArrayList and then uses this list collection object’s reference to add a pair of Employee objects to the list. It then adds a String object, which violates the implied contract that ArrayList is supposed to store only Employee objects.

main() next obtains a java.util.Iterator instance for iterating over the list of Employees. As long as Iterator’s hasNext() method returns true, its next() method is called to return an object stored in the array list.

The Object that next() returns must be downcast to Employee so that the Employee object’s getName() method can be called to return the employee’s name. The string that this method returns is then output to the standard output device via System.out.println().

The (Employee) cast checks the type of each object returned by next() to make sure that it is an Employee. Although this is true of the first two objects, it’s not true of the third object. The attempt to cast "Jack Frost" to Employee results in a ClassCastException.

The ClassCastException occurs because of an assumption that a list is homogenous. In other words, a list stores only objects of a single type or a family of related types. In reality, the list is heterogeneous in that it can store any Object.

Listing 6-21’s generics-based homogenous list avoids ClassCastException.

Listing 6-21. Lack of Type Safety Leading to a Compiler Error

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```
class Employee
{
    private String name;
    
    Employee(String name)
    {
        this.name = name;
    }
    
    String getName()
    {
        return name;
    }
}

public class TypeSafety
{
    public static void main(String[] args)
    {
        List<Employee> employees = new ArrayList<Employee>();
        employees.add(new Employee("John Doe");
        employees.add(new Employee("Jane Smith");
        employees.add("Jack Frost");
        Iterator<Employee> iter = employees.iterator();
        while (iter.hasNext())
        {
            Employee emp = iter.next();
            System.out.println(emp.getName());
        }
    }
}

Listing 6-21’s refactored main() method illustrates the central feature of generics, which is the parameterized type (a class or interface name followed by an angle-bracket delimited type list identifying what kinds of objects are legal in that context).

For example, java.util.List<Employee> indicates only Employee objects can be stored in the List. As shown, the <Employee> designation must be repeated with ArrayList, as in ArrayList<Employee>, which is the collection implementation that stores the Employees.

Also, Iterator<Employee> indicates that iterator() returns an Iterator whose next() method returns only Employee objects. It’s not necessary to cast iter.next()’s returned value to Employee because the compiler inserts the cast on your behalf.

If you attempt to compile this listing, the compiler will report an error when it encounters employees.add("Jack Frost");. The error message will tell you that the compiler cannot find an add(java.lang.String) method in the java.util.List<Employee> interface.

Unlike in the pre-generics List interface, which declares an add(Object) method, the generified List interface’s add() method parameter reflects the interface’s parameterized type name. For example, List<Employee> implies add(Employee).
Listing 6-20 reveals that the unsafe code causing the `ClassCastException` (`employees.add("Jack Frost");`) and the code that triggers the exception `((Employee) iter.next())` are quite close. However, they are often farther apart in larger applications.

Rather than having to deal with angry clients while hunting down the unsafe code that ultimately led to the `ClassCastException`, you can rely on the compiler saving you this frustration and effort by reporting an error when it detects this code during compilation. *Detecting type safety violations at compile time is the main benefit of using generics.*

**Generic Types**

A *generic type* is a class or interface that introduces a family of parameterized types by declaring a *formal type parameter list* (a comma-separated list of *type parameter* names between angle brackets). This syntax is expressed as follows:

```java
class identifier<formal_type_parameter_list> {}
interface identifier<formal_type_parameter_list> {}
```

For example, `List<E>` is a generic type, where `List` is an interface and type parameter `E` identifies the list's element type. Similarly, `Map<K, V>` is a generic type, where `Map` is an interface and type parameters `K` and `V` identify the map's key and value types.

*Note* When declaring a generic type, it's conventional to specify single uppercase letters as type parameter names. Furthermore, these names should be meaningful. For example, `E` indicates element, `T` indicates type, `K` indicates key, and `V` indicates value. If possible, you should avoid choosing a type parameter name that is meaningless where it is used. For example, `List<E>` means list of elements, but what does `List<S>` mean?

Parameterized types are instances of generic types. Each parameterized type replaces the generic type's type parameters with type names. For example, `List<Employee>` (List of Employee) and `List<String>` (List of String) are examples of parameterized types based on `List<E>`. Similarly, `Map<String, Employee>` is an example of a parameterized type based on `Map<K, V>`.

The type name that replaces a type parameter is known as an *actual type argument*. Five kinds of actual type arguments are supported by generics:

- **Concrete type**: The name of a class or interface is passed to the type parameter. For example, `List<Employee> employees;` specifies that the list elements are `Employee` instances.
- **Concrete parameterized type**: The name of a parameterized type is passed to the type parameter. For example, `List<List<String>> nameLists;` specifies that the list elements are lists of strings.
- **Array type**: An array is passed to the type parameter. For example, `List<String[]> countries;` specifies that the list elements are arrays of `Strings`, possibly city names.
Type parameter: A type parameter is passed to the type parameter. For example, given class declaration `class X<E> { List<E> queue; }`, X's type parameter E is passed to List's type parameter E.

Wildcard: The ? is passed to the type parameter. For example, `List<?> list;` specifies that the list elements are unknown. You will learn about wildcards later in this chapter.

A generic type also identifies a raw type, which is a generic type without its type parameters. For example, `List<Employee>`'s raw type is `List`. Raw types are nongeneric and can hold any `Object`.

Note: Java allows raw types to be intermixed with generic types to support the vast amount of legacy code that was written prior to the arrival of generics. However, the compiler outputs a warning message whenever it encounters a raw type in source code.

Declaring and Using Your Own Generic Types

It’s not difficult to declare your own generic types. In addition to specifying a formal type parameter list, your generic type specifies its type parameter(s) throughout its implementation. For example, Listing 6-22 declares a `Queue<E>` generic type.

Listing 6-22. Declaring and Using a `Queue<E>` Generic Type

```java
public class Queue<E>
{
    private E[] elements;
    private int head, tail;

    @SuppressWarnings("unchecked")
    Queue(int size)
    {
        if (size < 2)
            throw new IllegalArgumentException("" + size);
        elements = (E[]) new Object[size];
        head = 0;
        tail = 0;
    }

    void insert(E element) throws QueueFullException
    {
        if (isFull())
            throw new QueueFullException();
        elements[tail] = element;
        tail = (tail + 1) % elements.length;
    }
```
E remove() throws QueueEmptyException
{
    if (isEmpty())
        throw new QueueEmptyException();
    E element = elements[head];
    head = (head + 1) % elements.length;
    return element;
}

boolean isEmpty()
{
    return head == tail;
}

boolean isFull()
{
    return (tail + 1) % elements.length == head;
}

public static void main(String[] args)
    throws QueueFullException, QueueEmptyException
{
    Queue<String> queue = new Queue<String>(6);
    System.out.println("Empty: "+ queue.isEmpty());
    System.out.println("Full: "+ queue.isFull());
    System.out.println("Adding A");
    queue.insert("A");
    System.out.println("Adding B");
    queue.insert("B");
    System.out.println("Adding C");
    queue.insert("C");
    System.out.println("Adding D");
    queue.insert("D");
    System.out.println("Adding E");
    queue.insert("E");
    System.out.println("Empty: "+ queue.isEmpty());
    System.out.println("Full: "+ queue.isFull());
    System.out.println("Removing "+ queue.remove());
    System.out.println("Empty: "+ queue.isEmpty());
    System.out.println("Full: "+ queue.isFull());
    System.out.println("Adding F");
    queue.insert("F");
    while (!queue.isEmpty())
        System.out.println("Removing "+ queue.remove());
    System.out.println("Empty: "+ queue.isEmpty());
    System.out.println("Full: "+ queue.isFull());
}
class QueueEmptyException extends Exception
{
}

class QueueFullException extends Exception
{
}

Listing 6-22 declares Queue, QueueEmptyException, and QueueFullException classes. The latter two classes describe checked exceptions that are thrown from methods of the former class.

Queue implements a **queue**, a data structure that stores elements in first-in, first-out order. An element is inserted at the **tail** and removed at the **head**. The queue is empty when the head equals the tail and full when the tail is one less than the head. As a result, a queue of size n can store a maximum of n - 1 elements.

Notice that Queue\<E>’s E type parameter appears throughout the source code. For example, E appears in the elements array declaration to denote the array’s element type. E is also specified as the type of insert()’s parameter and as remove()’s return type.

E also appears in elements = (E[]) new Object[size]; (I will explain later why I specified this expression instead of specifying the more compact elements = new E[size]; expression.)

The E[] cast results in the compiler warning about this cast being unchecked. The compiler is concerned that downcasting from Object[] to E[] might result in a violation of type safety because any kind of object can be stored in Object[].

The compiler’s concern isn’t justified in this example. There is no way that a non-E object can appear in the E[] array. Because the warning is meaningless in this context, it is suppressed by prefixing the constructor with @SuppressWarnings("unchecked").

**Caution** Be careful when suppressing an unchecked warning. You must first prove that a ClassCastException cannot occur, and then you can suppress the warning.

When you run this application, it generates the following output:

```
Empty: true
Full: false
Adding A
Adding B
Adding C
Adding D
Adding E
Empty: false
Full: true
Removing A
Empty: false
Full: false
```
Type Parameter Bounds

List\(<E>\)'s E type parameter and Map\(<K, \ V>\)'s K and V type parameters are examples of unbounded type parameters. You can pass any actual type argument to an unbounded type parameter.

It is sometimes necessary to restrict the kinds of actual type arguments that can be passed to a type parameter. For example, you might want to declare a class whose instances can only store instances of classes that subclass an abstract Shape class (such as Circle and Rectangle).

To restrict actual type arguments, you can specify an upper bound, a type that serves as an upper limit on the types that can be chosen as actual type arguments. The upper bound is specified via reserved word extends followed by a type name.

For example, ShapesList\(<E \ extends \ Shape>\) identifies Shape as an upper bound. You can specify ShapesList\(<Circle>\), ShapesList\(<Rectangle>\), and even ShapesList\(<Shape>\), but not ShapesList\(<String>\) because String is not a subclass of Shape.

You can assign more than one upper bound to a type parameter, where the first bound is a class or interface and where each additional upper bound is an interface, by using the ampersand character (\&\&) to separate bound names. Consider Listing 6-23.

Listing 6-23. Assigning Multiple Upper Bounds to a Type Parameter

```java
abstract class Shape
{
}

class Circle extends Shape implements Comparable<Circle>
{
    private double x, y, radius;

    Circle(double x, double y, double radius)
    {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public int compareTo(Circle circle)
    {
        if (radius < circle.radius)
            return -1;
    }
```
else
    if (radius > circle.radius)
        return 1;
    else
        return 0;
}

@Override
public String toString()
{
    return "(" + x + ", " + y + ", " + radius + ")";
}

class SortedShapesList<S extends Shape & Comparable<S>>
{
    @SuppressWarnings("unchecked")
    private S[] shapes = (S[]) new Shape[2];
    private int index = 0;

    void add(S shape)
    {
        shapes[index++] = shape;
        if (index < 2)
            return;
        System.out.println("Before sort: " + this);
        sort();
        System.out.println("After sort: " + this);
    }

    private void sort()
    {
        if (index == 1)
            return;
        if (shapes[0].compareTo(shapes[1]) > 0)
        {
            S shape = (S) shapes[0];
            shapes[0] = shapes[1];
            shapes[1] = shape;
        }
    }

    @Override
    public String toString()
    {
        return shapes[0].toString() + " " + shapes[1].toString();
    }
}
public class SortedShapesListDemo
{
    public static void main(String[] args)
    {
        SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();
        ssl.add(new Circle(100, 200, 300));
        ssl.add(new Circle(10, 20, 30));
    }
}

Listing 6-23's Circle class extends Shape and implements the java.lang.Comparable interface, which is used to specify the natural ordering of Circle objects. The interface's compareTo() method implements this ordering by returning a value to reflect the order.

- A negative value is returned when the current object should precede the object passed to compareTo() in some fashion.
- A zero value is returned when the current and argument objects are the same.
- A positive value is returned when the current object should succeed the argument object.

Circle's overriding compareTo() method compares two Circle objects based on their radii. This method orders a Circle instance with the smaller radius before a Circle instance with a larger radius.

The SortedShapesList class specifies <S extends Shape & Comparable<S>> as its parameter list. The actual type argument passed to the S parameter must subclass Shape, and it must also implement the Comparable interface.

Note: A type parameter bound that includes the type parameter is known as a recursive type bound. For example, Comparable<S> in S extends Shape & Comparable<S> is a recursive type bound. Recursive type bounds are rare and typically show up in conjunction with the Comparable interface for specifying a type's natural ordering.

Circle satisfies both criteria: it subclasses Shape and implements Comparable. As a result, the compiler doesn't report an error when it encounters the main() method's SortedShapesList<Circle> ssl = new SortedShapesList<Circle>(); statement.

An upper bound offers extra static type checking that guarantees that a parameterized type adheres to its bounds. This assurance means that the upper bound's methods can be called safely. For example, sort() can call Comparable's compareTo() method.

If you run this application, you will discover the following output, which shows that the two Circle objects are sorted in ascending order of radius:

Before sort: (100.0, 200.0, 300.0) (10.0, 20.0, 30.0)
After sort: (10.0, 20.0, 30.0) (100.0, 200.0, 300.0)
CHAPTER 6: Mastering Advanced Language Features, Part 2

Note Type parameters cannot have lower bounds. Angelika Langer explains the rationale for this restriction in her “Java Generics FAQs” at www.angelikalanger.com/GenericsFAQ/FAQSections/TypeParameters.html#FAQ107.

Type Parameter Scope

A type parameter’s scope (visibility) is its generic type except where masked (hidden). This scope includes the formal type parameter list of which the type parameter is a member. For example, the scope of S in SortedShapesList<S extends Shape & Comparable<S>> is all of SortedShapesList and the formal type parameter list.

It is possible to mask a type parameter by declaring a same-named type parameter in a nested type’s formal type parameter list. For example, Listing 6-24 masks an enclosing class’s T type parameter.

Listing 6-24. Masking a Type Parameter

class EnclosingClass<T>
{
    static class EnclosedClass<T extends Comparable<T>>
    {
    }
}

EnclosingClass’s T type parameter is masked by EnclosedClass’s T type parameter, which specifies an upper bound where only those types that implement the Comparable interface can be passed to EnclosedClass. Referencing T from within EnclosedClass refers to the bounded T and not the unbounded T passed to EnclosingClass.

If masking is undesirable, it is best to choose a different name for the type parameter. For example, you might specify EnclosedClass<U extends Comparable<U>>. Although U is not as meaningful a name as T, this situation justifies this choice.

The Need for Wildcards

Suppose that you have created a List of String and want to output this list. Because you might create a List of Employee and other kinds of lists, you want this method to output an arbitrary List of Object. You end up creating Listing 6-25.

Listing 6-25. Attempting to Output a List of Object

import java.util.ArrayList;
import java.util.List;

public class OutputList
{
    public static void main(String[] args)
Now that you’ve accomplished your objective (or so you think), you compile Listing 6-25 via `javac OutputList.java`. Much to your surprise, you receive the following error message:

```
OutputList.java:12: error: method outputList in class OutputList cannot be applied to given types;
    outputList(ls);
^
  required: List<Object>
  found: List<String>
  reason: actual argument List<String> cannot be converted to List<Object> by method invocation conversion

1 error
```

This error message results from being unaware of the fundamental rule of generic types: **for a given subtype x of type y, and given G as a raw type declaration, G<x> is not a subtype of G<y>**.

To understand this rule, you must refresh your understanding of subtype polymorphism (see Chapter 4). Basically, a subtype is a specialized kind of its supertype. For example, `Circle` is a specialized kind of `Shape` and `String` is a specialized kind of `Object`. This polymorphic behavior also applies to related parameterized types with the same type parameters. For example, `List<Object>` is a specialized kind of `java.util.Collection<Object>`.

However, this polymorphic behavior doesn’t apply to multiple parameterized types that differ only in regard to one type parameter being a subtype of another type parameter. For example, `List<String>` is not a specialized kind of `List<Object>`.

The following example reveals why parameterized types differing only in type parameters are not polymorphic:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Employee());
String s = ls.get(0);
```

This example will not compile because it violates type safety. If it compiled, a `ClassCastException` instance would be thrown at runtime because of the implicit cast to `String` on the final line.
The first line instantiates a `List` of `String` and the second line upcasts its reference to a `List` of `Object`. The third line adds a new `Employee` object to the `List` of `Object`. The fourth line obtains the `Employee` object via `get()` and attempts to assign it to the `List` of `String` reference variable. However, `ClassCastException` is thrown because of the implicit cast to `String`—an `Employee` is not a `String`.

**Note** Although you cannot upcast `List<String>` to `List<Object>`, you can upcast `List<String>` to the raw type `List` in order to interoperate with legacy code.

The aforementioned error message reveals that `List` of `String` is not also `List` of `Object`. To call Listing 6-25's `outputList()` method without violating type safety, you can only pass an argument of `List<Object>` type, which limits the usefulness of this method.

However, generics offer a solution: the wildcard argument (`?`), which stands for any type. By changing `outputList()`'s parameter type from `List<Object>` to `List<?>`, you can call `outputList()` with a `List` of `String`, a `List` of `Employee`, and so on.

**Generic Methods**

Suppose you need a method to copy a `List` of any kind of object to another `List`. Although you might consider coding a `void copyList(List<Object> src, List<Object> dest)` method, this method would have limited usefulness because it could only copy lists whose element type is `Object`. You couldn't copy a `List<Employee>`, for example.

If you want to pass source and destination lists whose elements are of arbitrary type (but their element types agree), you need to specify the wildcard character as a placeholder for that type. For example, you might consider writing the following `copyList()` class method that accepts collections of arbitrary-typed objects as its arguments:

```java
static void copyList(List<?> src, List<?> dest)
{
    for (int i = 0; i < src.size(); i++)
        dest.add(src.get(i));
}
```

This method's parameter list is correct, but there is another problem: the compiler outputs the following error message when it encounters `dest.add(src.get(i));`:

```
CopyList.java:19: error: no suitable method found for add(Object)
    dest.add(src.get(i));
^  
  method List.add(int,CAP#1) is not applicable
    (actual and formal argument lists differ in length)
  method List.add(CAP#1) is not applicable
    (actual argument Object cannot be converted to CAP#1 by method invocation conversion)
where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```
This error message assumes that `copyList()` is part of a class named `CopyList`. Although it appears to be incomprehensible, the message basically means that the `dest.add(src.get(i))` method call violates type safety. Because `?` implies that any type of object can serve as a list’s element type, it’s possible that the destination list’s element type is incompatible with the source list’s element type.

For example, suppose you create a `List` of `String` as the source list and a `List` of `Employee` as the destination list. Attempting to add the source list’s elements to the destination list, which expects `Employee`, violates type safety. If this copy operation were allowed, a `ClassCastException` instance would be thrown when trying to obtain the destination list’s elements.

You could solve this problem in a limited way as follows:

```java
static void copyList(List<? extends String> src,
                     List<? super String> dest)
{
    for (int i = 0; i < src.size(); i++)
        dest.add(src.get(i));
}
```

This method demonstrates a wildcard argument feature in which you can supply an upper bound or (unlike with a type parameter) a lower bound to limit the types that can be passed as actual type arguments to the generic type. Specifically, it shows an upper bound via `extends` followed by the upper bound type after the `?`, and a lower bound via `super` followed by the lower bound type after the `?`.

You interpret `? extends String` to mean that any actual type argument that is `String` or a subclass of this type can be passed, and you interpret `? super String` to imply that any actual type argument that is `String` or a superclass of this type can be passed. Because `String` cannot be subclassed, this means that you can only pass source lists of `String` and destination lists of `String` or `Object`.

The problem of copying lists of arbitrary element types to other lists can be solved through the use of a *generic method* (a class or instance method with a type-generalized implementation). Generic methods are syntactically expressed as follows:

```java
<formal_type_parameter_list> return_type identifier(parameter_list)
```

The `formal_type_parameter_list` is the same as when specifying a generic type: it consists of type parameters with optional bounds. A type parameter can appear as the method’s `return_type`, and type parameters can appear in the `parameter_list`. The compiler infers the actual type arguments from the context in which the method is invoked.

You’ll discover many examples of generic methods in the Collections Framework. For example, its `java.util.Collections` class provides a `public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)` method for returning the minimum element in the given collection according to the natural ordering of its elements.

You can easily convert `copyList()` into a generic method by prefixing the return type with `<T>` and replacing each wildcard with `T`. The resulting method header is `<T> void copyList(List<T> src, List<T> dest)`, and Listing 6-26 presents its source code as part of an application that copies a `List` of `Circle` to another `List` of `Circle`.  

www.it-ebooks.info
import java.util.ArrayList;
import java.util.List;

class Circle {
    private double x, y, radius;

    Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ", " + radius + ")";
    }
}

public class CopyList {
    public static void main(String[] args) {
        List<String> ls = new ArrayList<String>();
        ls.add("A");
        ls.add("B");
        ls.add("C");
        outputList(ls);
        List<String> lsCopy = new ArrayList<String>();
        copyList(ls, lsCopy);
        outputList(lsCopy);
        List<Circle> lc = new ArrayList<Circle>();
        lc.add(new Circle(10.0, 20.0, 30.0));
        lc.add(new Circle (5.0, 4.0, 16.0));
        outputList(lc);
        List<Circle> lcCopy = new ArrayList<Circle>();
        copyList(lc, lcCopy);
        outputList(lcCopy);
    }

    static <T> void copyList(List<T> src, List<T> dest) {
        for (int i = 0; i < src.size(); i++)
            dest.add(src.get(i));
    }
}
static void outputList(List<?> list)
{
    for (int i = 0; i < list.size(); i++)
        System.out.println(list.get(i));
    System.out.println();
}

The compiler uses a type inference algorithm to infer a generic method's type arguments from
the context in which the method was invoked. For example, the compiler determines that
copyList(ls, lsCopy); copies a List of String to another List of String. Similarly, it determines
that copyList(lc, lCCopy); copies a List of Circle to another List of Circle. Without this algorithm,
you would have to specify these arguments, as in CopyList.<String>copyList(ls, lsCopy); and
CopyList.<Circle>copyList(lc, lCCopy);

Compile Listing 6-26 (javac CopyList.java) and run this application (java CopyList). You should
observe the following output:

A
B
C

A
B
C

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)
Arrays and Generics

After presenting Listing 6-22’s `Queue<E>` generic type, I mentioned that I would explain why I specified `elements = (E[]) new Object[size];` instead of the more compact `elements = new E[size];` expression. Because of Java’s generics implementation, it isn’t possible to specify array-creation expressions that involve type parameters (such as `new E[size]` or `new List<E>[50]`) or actual type arguments (such as `new Queue<String>[15]`). If you attempt to do so, the compiler will report a generic array creation error message.

Before I present an example that demonstrates why allowing array-creation expressions that involve type parameters or actual type arguments is dangerous, you need to understand reification and covariance in the context of arrays, and erasure, which is at the heart of how generics are implemented.

Reification is representing the abstract as if it was concrete—for example, making a memory address available for direct manipulation by other language constructs. Java arrays are reified in that they’re aware of their element types (an element type is stored internally) and can enforce these types at runtime. Attempting to store an invalid element in an array causes the virtual machine to throw an instance of the `java.lang.ArrayStoreException` class.

Listing 6-27 teaches you how array manipulation can lead to an `ArrayStoreException`.

**Listing 6-27. How an ArrayStoreException Arises**

```java
class Point
{
    int x, y;
}

class ColoredPoint extends Point
{
    int color;
}
```
public class ReificationDemo
{
    public static void main(String[] args)
    {
        ColoredPoint[] cptArray = new ColoredPoint[1];
        Point[] ptArray = cptArray;
        ptArray[0] = new Point();
    }
}

Listing 6-27’s main() method first instantiates a ColoredPoint array that can store one element. In contrast to this legal assignment (the types are compatible), specifying ColoredPoint[] cptArray = new Point[1]; is illegal (and won’t compile) because it would result in a ClassCastException at runtime—the array knows that the assignment is illegal.

Note If it’s not obvious, ColoredPoint[] cptArray = new Point[1]; is illegal because Point instances have fewer members (only x and y) than ColoredPoint instances (x, y, and color). Attempting to access a Point instance’s nonexistent color field from its entry in the ColoredPoint array would result in a memory violation (because no memory has been assigned to color) and ultimately crash the virtual machine.

The second line (Point[] ptArray = cptArray;) is legal because of covariance (an array of supertype references is a supertype of an array of subtype references). In this case, an array of Point references is a supertype of an array of ColoredPoint references. The nonarray analogy is that a subtype is also a supertype. For example, a java.lang.Throwable instance is a kind of Object instance.

Covariance is dangerous when abused. For example, the third line (ptArray[0] = new Point();) results in ArrayStoreException at runtime because a Point instance is not a ColoredPoint instance. Without this exception, an attempt to access the nonexistent member color crashes the virtual machine.

Unlike with arrays, a generic type’s type parameters are not reified. They’re not available at runtime because they’re thrown away after the source code is compiled. This “throwing away of type parameters” is a result of erasure, which also involves inserting casts to appropriate types when the code isn’t type correct, and replacing type parameters by their upper bounds (such as Object).

Note The compiler performs erasure to let generic code interoperate with legacy (nongeneric) code. It transforms generic source code into nongeneric runtime code. One consequence of erasure is that you cannot use the instanceof operator with parameterized types apart from unbounded wildcard types. For example, it’s illegal to specify List<Employee> le = null; if (le instanceof ArrayList<Employee>) {}. Instead, you must change the instanceof expression to le instanceof ArrayList<?> (unbounded wildcard) or le instanceof ArrayList (raw type, which is the preferred use).
Suppose you could specify an array-creation expression involving a type parameter or an actual type argument. Why would this be bad? For an answer, consider the following example, which should generate an ArrayStoreException instead of a ClassCastException but doesn’t do so:

```
List<Employee>[] empListArray = new ArrayList<Employee>[1];
List<String> strList = new ArrayList<String>();
strList.add("string");
Object[] objArray = empListArray;
objArray[0] = strList;
Employee e = empListArray[0].get(0);
```

Assume that the first line, which creates a one-element array where this element stores a List of Employee, is legal. The second line creates a List of String, and the third line stores a single String object in this list.

The fourth line assigns `empListArray` to `objArray`. This assignment is legal because arrays are covariant and erasure converts `List<Employee>[]` to the `List` runtime type and `List` subtypes `Object`.

Because of erasure, the virtual machine doesn’t throw `ArrayStoreException` when it encounters `objArray[0] = strList;`. After all, you’re assigning a `List` reference to a `List[]` array at runtime. However, this exception would be thrown if generic types were reified because you’d then be assigning a `List<String>` reference to a `List<Employee>[]` array.

However, there is a problem. A `List<String>` instance has been stored in an array that can only hold `List<Employee>` instances. When the compiler-inserted cast operator attempts to cast `empListArray[0].get(0)`’s return value ("string") to `Employee`, the cast operator throws a `ClassCastException` object.

### Mastering Enums

An enumerated type is a type that specifies a named sequence of related constants as its legal values. The months in a calendar, the coins in a currency, and the days of the week are examples of enumerated types.

Java developers have traditionally used sets of named integer constants to represent enumerated types. Because this form of representation has proven to be problematic, Java 5 introduced the `enum` alternative.

In this section, I will introduce you to `enum`. After discussing the problems with traditional enumerated types, I will present the `enum` alternative. I will then introduce you to the `Enum` class, from which `enum` originate.

### The Trouble with Traditional Enumerated Types

Listing 6-28 declares a `Coin` enumerated type whose set of constants identifies different kinds of coins in a currency.
Listing 6-28. An Enumerated Type Identifying Coins

class Coin
{
    final static int PENNY = 0;
    final static int NICKEL = 1;
    final static int DIME = 2;
    final static int QUARTER = 3;
}

Listing 6-29 declares a Weekday enumerated type whose constants identify the days of the week.

Listing 6-29. An Enumerated Type Identifying Weekdays

class Weekday
{
    final static int SUNDAY = 0;
    final static int MONDAY = 1;
    final static int TUESDAY = 2;
    final static int WEDNESDAY = 3;
    final static int THURSDAY = 4;
    final static int FRIDAY = 5;
    final static int SATURDAY = 6;
}

Listing 6-28’s and 6-29’s approach to representing an enumerated type is problematic, where the biggest problem is the lack of compile-time type safety. For example, you can pass a coin to a method that requires a weekday and the compiler will not complain.

You can also compare coins to weekdays, as in Coin.NICKEL == Weekday.MONDAY, and specify even more meaningless expressions, such as Coin.DIME + Weekday.FRIDAY - 1 / Coin.QUARTER. The compiler doesn’t complain because it only sees ints.

Applications that depend upon enumerated types are brittle. Because the type’s constants are compiled into an application’s classfiles, changing a constant’s int value requires you to recompile dependent applications or risk them behaving erratically.

Another problem with enumerated types is that int constants cannot be translated into meaningful string descriptions. For example, what does the number 4 mean when debugging a faulty application? Being able to see THURSDAY instead of 4 would be more helpful.

Note You could circumvent the previous problem by using String constants. For example, you might specify public final static String THURSDAY = "THURSDAY"; Although the constant value is more meaningful, String-based constants can impact performance because you cannot use == to efficiently compare just any old strings (as you will discover in Chapter 7). Other problems related to String-based constants include hard-coding the constant’s value ("THURSDAY") instead of the constant’s name (THURSDAY) into source code, which makes it very difficult to change the constant’s value at a later time; and misspelling a hard-coded constant ("THURZDAY"), which compiles correctly but is problematic at runtime.
The Enum Alternative

Java 5 introduced enums as a better alternative to traditional enumerated types. An enum is an enumerated type that is expressed via reserved word enum. The following example uses enum to declare Listing 6-28’s and 6-29’s enumerated types:

```java
enum Coin { PENNY, NICKEL, DIME, QUARTER }
enum Weekday { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

Despite their similarity to the int-based enumerated types found in C++ and other languages, this example’s enums are classes. Each constant is a public static final field that represents an instance of its enum class.

Because constants are final, and because you cannot call an enum’s constructors to create more constants, you can use == to compare constants efficiently and (unlike string constant comparisons) safely. For example, you can specify `c == Coin.NICKEL`.

 Enums promote compile-time type safety by preventing you from comparing constants in different enums. For example, the compiler will report an error when it encounters `Coin.PENNY == Weekday.SUNDAY`.

The compiler also frowns on passing a constant of the wrong enum kind to a method. For example, you cannot pass `Weekday.FRIDAY` to a method whose parameter type is `Coin`.

Applications depending on enums are not brittle because the enum’s constants are not compiled into an application’s classfiles. Also, the enum provides a `toString()` method for returning a more useful description of a constant’s value.

Because enums are so useful, Java 5 enhanced the switch statement to support them. Listing 6-30 demonstrates this statement switching on one of the constants in the previous example’s `Coin` enum.

Listing 6-30. Using the Switch Statement with an Enum

```java
public class EnhancedSwitch {
    enum Coin { PENNY, NICKEL, DIME, QUARTER }

    public static void main(String[] args) {
        Coin coin = Coin.NICKEL;
        switch (coin) {
            case PENNY : System.out.println("1 cent"); break;
            case NICKEL : System.out.println("5 cents"); break;
            case DIME   : System.out.println("10 cents"); break;
            case QUARTER: System.out.println("25 cents"); break;
            default     : assert false;
        }
    }
}
```
Listing 6-30 demonstrates switching on an enum’s constants. This enhanced statement only allows you to specify the name of a constant as a case label. If you prefix the name with the enum, as in `case Coin.DIME`, the compiler reports an error.

**Enhancing an Enum**

You can add fields, constructors, and methods to an enum—you can even have the enum implement interfaces. For example, Listing 6-31 adds a field, a constructor, and two methods to `Coin` to associate a denomination value with a `Coin` constant (such as 1 for penny and 5 for nickel) and convert pennies to the denomination.

*Listing 6-31. Enhancing the `Coin` Enum*

```java
enum Coin {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25);

    private final int denomValue;

    Coin(int denomValue) {
        this.denomValue = denomValue;
    }

    int denomValue() {
        return denomValue;
    }

    int toDenomination(int numPennies) {
        return numPennies / denomValue;
    }
}
```

Listing 6-31’s constructor accepts a denomination value, which it assigns to a private blank final field named `denomValue`—all fields should be declared `final` because constants are immutable. Notice that this value is passed to each constant during its creation (`PENNY(1)`, for example).

**Caution** When the comma-separated list of constants is followed by anything other than an enum’s closing brace, you must terminate the list with a semicolon or the compiler will report an error.
Furthermore, this listing’s `denomValue()` method returns `denomValue`, and its `toDenomination()` method returns the number of coins of that denomination that are contained within the number of pennies passed to this method as its argument. For example, 3 nickels are contained in 16 pennies.

Listing 6-32 shows how to use the enhanced Coin enum.

**Listing 6-32. Exercising the Enhanced Coin Enum**

```java
class Coins {
    public static void main(String[] args) {
        if (args.length == 1) {
            int numPennies = Integer.parseInt(args[0]);
            System.out.println(numPennies + " pennies is equivalent to: ");
            int numQuarters = Coin.QUARTER.toDenomination(numPennies);
            System.out.println(numQuarters + " " + Coin.QUARTER.toString() +
            (numQuarters != 1 ? "s, " : ", "));
            numPennies -= numQuarters * Coin.QUARTER.denomValue();
            int numDimes = Coin.DIME.toDenomination(numPennies);
            System.out.println(numDimes + " " + Coin.DIME.toString() +
            (numDimes != 1 ? "s, " : ", "));
            numPennies -= numDimes * Coin.DIME.denomValue();
            int numNickels = Coin.NICKEL.toDenomination(numPennies);
            System.out.println(numNickels + " " + Coin.NICKEL.toString() +
            (numNickels != 1 ? "s, " : ", and"));
            numPennies -= numNickels * Coin.NICKEL.denomValue();
            System.out.println(numPennies + " " + Coin.PENNY.toString() +
            (numPennies != 1 ? "s" : "");
        }
        System.out.println();
        System.out.println("Denomination values:");
        for (int i = 0; i < Coin.values().length; i++)
            System.out.println(Coin.values()[i].denomValue());
    }
}
```

Listing 6-32 describes an application that converts its solitary “pennies” command-line argument to an equivalent amount expressed in quarters, dimes, nickels, and pennies. In addition to calling a Coin constant’s `denomValue()` and `toDenomination()` methods, the application calls `toString()` to output a string representation of the coin.

Another called enum method is `values()`. This method returns an array of all Coin constants that are declared in the Coin enum (value()’s return type, in this example, is Coin[]). This array is useful when you need to iterate over these constants. For example, Listing 6-32 calls this method to output each coin's denomination.
When you run this application with 119 as its command-line argument (java Coins 119), it generates the following output:

119 pennies is equivalent to:
4 QUARTERs,
1 DIME,
1 NICKEL, and
4 PENNYs

Denomination values:
1
5
10
25

The output shows that toString() returns a constant’s name. It is sometimes useful to override this method to return a more meaningful value. For example, a method that extracts tokens (named character sequences) from a string might use a Token enum to list token names and, via an overriding toString() method, values—see Listing 6-33.

**Listing 6-33. Overriding toString() to Return a Token Constant’s Value**

```java
public enum Token {
    IDENTIFIER("ID"),
    INTEGER("INT"),
    LPAREN("(") ,
    RPAREN(")" ),
    COMMA(",");

    private final String tokValue;

    Token(String tokValue)
    {
        this.tokValue = tokValue;
    }

    @Override
    public String toString()
    {
        return tokValue;
    }

    public static void main(String[] args)
    {
        System.out.println("Token values:");
        for (int i = 0; i < Token.values().length; i++)
            System.out.println(Token.values()[i].name() + " = " +
                                Token.values()[i]);
    }
}
```

www.it-ebooks.info
Listing 6-33's `main()` method calls `values()` to return the array of `Token` constants. For each constant, it calls the constant's `name()` method to return the constant's name, and implicitly calls `toString()` to return the constant's value. If you were to run this application, you would observe the following output:

Token values:
IDENTIFIER = ID
INTEGER = INT
LPAREN = (
RPAREN = )
COMMA = ,

Another way to enhance an enum is to assign a different behavior to each constant. You can accomplish this task by introducing an abstract method into the enum and overriding this method in an anonymous subclass of the constant. Listing 6-34's `TempConversion` enum demonstrates this technique.

Listing 6-34. Using Anonymous Subclasses to Vary the Behaviors of Enum Constants

```java
public enum TempConversion
{
    C2F("Celsius to Fahrenheit")
    {
        @Override
        double convert(double value)
        {
            return value * 9.0 / 5.0 + 32.0;
        }
    },
    F2C("Fahrenheit to Celsius")
    {
        @Override
        double convert(double value)
        {
            return (value - 32.0) * 5.0 / 9.0;
        }
    },
    TempConversion(String desc)
    {
        this.desc = desc;
    }
    private String desc;
    @Override
    public String toString()
    {
        return desc;
    }
}
```
abstract double convert(double value);

public static void main(String[] args)
{
    System.out.println(C2F + " for 100.0 degrees = " + C2F.convert(100.0));
    System.out.println(F2C + " for 98.6 degrees = " + F2C.convert(98.6));
}

When you run this application, it generates the following output:

Celsius to Fahrenheit for 100.0 degrees = 212.0
Fahrenheit to Celsius for 98.6 degrees = 37.0

The Enum Class

The compiler regards enum as syntactic sugar. When it encounters an enum type declaration (enum Coin {}), it generates a class whose name (Coin) is specified by the declaration, which also subclasses the abstract Enum class (in the java.lang package), the common base class of all Java language-based enumeration types.

If you examine Enum's Java documentation, you will discover that it overrides Object's clone(), equals(), finalize(), hashCode(), and toString() methods.

- clone() is overridden to prevent constants from being cloned so that there is never more than one copy of a constant; otherwise, constants could not be compared via ==.
- equals() is overridden to compare constants via their references—constants with the same identities (==) must have the same contents (equals()), and different identities imply different contents.
- finalize() is overridden to ensure that constants cannot be finalized.
- hashCode() is overridden because equals() is overridden.
- toString() is overridden to return the constant's name.

Except for toString(), all of the overriding methods are declared final so that they cannot be overridden in a subclass.

Enum also provides its own methods. These methods include the final compareTo(), (Enum implements Comparable), getDeclaringClass(), name(), and ordinal() methods.

- compareTo() compares the current constant with the constant passed as an argument to see which constant precedes the other constant in the enum and returns a value indicating their order. This method makes it possible to sort an array of unsorted constants.
- getDeclaringClass() returns the Class object corresponding to the current constant's enum. For example, the Class object for Coin is returned when calling Coin.PENNY.getDeclaringClass() for enum Coin { PENNY, NICKEL, DIME, QUARTER}. Also, TempConversion is returned when calling TempConversion.C2F. getDeclaringClass() for Listing 6-34's TempConversion enum. The compareTo()}
The method uses `Class`'s `getClass()` method and `Enum`'s `getDeclaringClass()` method to ensure that only constants belonging to the same enum are compared. Otherwise, a `ClassCastException` is thrown.

- name() returns the constant's name. Unless overridden to return something more descriptive, `toString()` also returns the constant's name.
- `ordinal()` returns a zero-based `ordinal`, an integer that identifies the position of the constant within the enum type. `compareTo()` compares ordinals.

Enum also provides the `public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)` method for returning the enum constant from the specified enum with the specified name.

- `enumType` identifies the `Class` object of the enum from which to return a constant.
- `name` identifies the name of the constant to return.

For example, `Coin penny = Enum.valueOf(Coin.class, "PENNY");` assigns the `Coin` constant whose name is `PENNY` to `penny`.

You will not discover a `values()` method in `Enum`'s Java documentation because the compiler synthesizes (manufactures) this method while generating the class.

### Extending the Enum Class

`Enum`'s generic type is `Enum<E extends Enum<E>>`. Although the formal type parameter list looks ghastly, it's not that hard to understand. But first, take a look at Listing 6-35.

#### Listing 6-35. The Coin Class As It Appears from the Perspective of Its Classfile

```java
public final class Coin extends Enum<Coin> {
    public static final Coin PENNY = new Coin("PENNY", 0);
    public static final Coin NICKEL = new Coin("NICKEL", 1);
    public static final Coin DIME = new Coin("DIME", 2);
    public static final Coin QUARTER = new Coin("QUARTER", 3);
    private static final Coin[] $VALUES = { PENNY, NICKEL, DIME, QUARTER };

    public static Coin[] values()
    {
        return Coin.$VALUES.clone();
    }

    public static Coin valueOf(String name)
    {
        return Enum.valueOf(Coin.class, "Coin");
    }

    private Coin(String name, int ordinal)
    {
        super(name, ordinal);
    }
}
```

www.it-ebooks.info
Behind the scenes, the compiler converts `enum Coin { PENNY, NICKEL, DIME, QUARTER}` into a class declaration that is similar to Listing 6-35.

The following rules show you how to interpret `Enum<E extends Enum<E>>` in the context of `Coin extends Enum<Coin>`:

- Any subclass of `Enum` must supply an actual type argument to `Enum`. For example, `Coin`'s header specifies `Enum<Coin>`.
- The actual type argument must be a subclass of `Enum`. For example, `Coin` is a subclass of `Enum`.
- A subclass of `Enum` (such as `Coin`) must follow the idiom that it supplies its own name (`Coin`) as an actual type argument.

The third rule allows `Enum` to declare methods—`compareTo()`, `getDeclaringClass()`, and `valueOf()`—whose parameter and/or return types are specified in terms of the subclass (`Coin`) and not in terms of `Enum`. The rationale for doing this is to avoid having to specify casts. For example, you don’t need to cast `valueOf()`’s return value to `Coin` in

```java
Coin penny = Enum.valueOf(Coin.class, "PENNY");
```

**Note** You cannot compile Listing 6-35 because the compiler will not compile any class that extends `Enum`. It will also complain about `super(name, ordinal);`.

## EXERCISES

The following exercises are designed to test your understanding of Chapter 6’s content.

1. What is an assertion?
2. When would you use assertions?
3. True or false: Specifying the `-ea` command-line option with no argument enables all assertions, including system assertions.
4. Define annotation.
5. What kinds of application elements can be annotated?
6. Identify the three compiler-supported annotation types.
7. How do you declare an annotation type?
8. What is a marker annotation?
9. What is an element?
10. How do you assign a default value to an element?
11. What is a meta-annotation?
12. Identify Java’s four meta-annotation types.
14. Why would you use generics?
15. What is the difference between a generic type and a parameterized type?
16. Which one of the nonstatic member class, local class, and anonymous class inner class categories cannot be generic?
17. Identify the five kinds of actual type arguments.
18. True or false: You cannot specify the name of a primitive type (such as double or int) as an actual type argument.
19. What is a raw type?
20. When does the compiler report an unchecked warning message and why?
21. How do you suppress an unchecked warning message?
22. True or false: List<E>'s E type parameter is unbounded.
23. How do you specify a single upper bound?
24. What is a recursive type bound?
25. Why are wildcard type arguments necessary?
26. What is a generic method?
27. In Listing 6-36, which overloaded method does the methodCaller() generic method call?

Listing 6-36. Which someOverloadedMethod() Is Called?

```java
import java.util.Date;

public class CallOverloadedNGMethodFromGMethod {
    public static void someOverloadedMethod(Object o) {
        System.out.println("call to someOverloadedMethod(Object o) ");
    }
    public static void someOverloadedMethod(Date d) {
        System.out.println("call to someOverloadedMethod(Date d) ");
    }
    public static <T> void methodCaller(T t) {
        someOverloadedMethod(t);
    }
    public static void main(String[] args) {
        methodCaller(new Date());
    }
}
```
28. What is reification?
29. True or false: Type parameters are reified.
30. What is erasure?
31. Define enumerated type.
32. Identify three problems that can arise when you use enumerated types whose constants are int-based.
33. What is an enum?
34. How do you use the switch statement with an enum?
35. In what ways can you enhance an enum?
36. What is the purpose of the abstract Enum class?
37. What is the difference between Enum's name() and toString() methods?
38. True or false: Enum's generic type is Enum<E extends Enum<E>>.
39. Declare a ToDo marker annotation type that annotates only type elements and that also uses the default retention policy.
40. Rewrite the StubFinder application to work with Listing 6-13's Stub annotation type (with appropriate @Target and @Retention annotations) and Listing 6-14's Deck class.
41. Implement a Stack<E> generic type in a manner that is similar to Listing 6-22's Queue class. Stack must declare push(), pop(), and isEmpty() methods (it could also declare an isFull() method, but that method is not necessary in this exercise); push() must throw a StackFullException instance when the stack is full; and pop() must throw a StackEmptyException instance when the stack is empty. (You must create your own StackFullException and StackEmptyException helper classes because they are not provided for you in the standard class library.) Declare a similar main() method, and insert two assertions into this method that validate your assumptions about the stack being empty immediately after being created and immediately after popping the last element.
42. Declare a Compass enum with NORTH, SOUTH, EAST, and WEST members. Declare a UseCompass class whose main() method randomly selects one of these constants and then switches on that constant. Each of the switch statement's cases should output a message such as heading north.

Summary

An assertion is a statement that lets you express an assumption of application correctness via a Boolean expression. If this expression evaluates to true, execution continues with the next statement. Otherwise, an error that identifies the cause of failure is thrown.

There are many situations where assertions should be used. These situations organize into internal invariant, control-flow invariant, and design-by-contract categories. An invariant is something that doesn't change.

Although there are many situations where assertions should be used, there also are situations where they should be avoided. For example, you should not use assertions to check the arguments that are passed to public methods.
The compiler records assertions in the classfile. However, assertions are disabled at runtime because they can affect performance. You must enable the classfile’s assertions before you can test assumptions about the behaviors of your classes.

Annotations are instances of annotation types and associate metadata with application elements. They are expressed in source code by prefixing their type names with @ symbols. For example, @Readonly is an annotation and Readonly is its type.

Java supplies a wide variety of annotation types, including the compiler-oriented Override, Deprecated, and SuppressWarnings types. However, you can also declare your own annotation types by using the @interface syntax.

Annotation types can be annotated with meta-annotations that identify the application elements they can target (such as constructors, methods, or fields), their retention policies, and other characteristics. Annotations whose types are assigned a runtime retention policy via @Retention annotations can be processed at runtime using custom applications. (Java 5 introduced an apt tool for this purpose, but its functionality was largely integrated into the compiler starting with Java 6.)

Java 5 introduced generics, language features for declaring and using type-agnostic classes and interfaces. While working with Java’s Collections Framework, these features help you avoid ClassCastExceptions.

A generic type is a class or interface that introduces a family of parameterized types by declaring a formal type parameter list. The type name that replaces a type parameter is known as an actual type argument.

There are five kinds of actual type arguments: concrete type, concrete parameterized type, array type, type parameter, and wildcard. Furthermore, a generic type also identifies a raw type, which is a generic type without its type parameters.

A generic method is a class or instance method with a type-generalized implementation, for example, <T> void copyList(List<T> src, List<T> dest). The compiler infers the actual type argument from the context in which the method is invoked.

An enumerated type is a type that specifies a named sequence of related constants as its legal values. Java developers have traditionally used sets of named integer constants to represent enumerated types.

Because sets of named integer constants have proven to be problematic, Java 5 introduced the enum alternative. An enum is an enumerated type that is expressed in source code via reserved word enum.

You can add fields, constructors, and methods to an enum—you can even have the enum implement interfaces. Also, you can override toString() to provide a more useful description of a constant’s value, and subclass constants to assign different behaviors.

The compiler regards enum as syntactic sugar for a class that subclasses Enum. This abstract class overrides various Object methods to provide default behaviors (usually for safety reasons), and provides additional methods for various purposes.

This chapter largely completes a tour of the Java language. In Chapter 7, I will begin to emphasize Java APIs by focusing on those basic APIs related to mathematics, string management, and more.
Chapter 7

Exploring the Basic APIs, Part 1

The standard class library’s java.lang and other packages provide many basic APIs that can benefit your Android apps. For example, you can perform mathematics operations and manipulate strings.

Exploring Math

The java.lang.Math class declares double constants E and PI that represent the natural logarithm base value (2.71828...) and the ratio of a circle’s circumference to its diameter (3.14159...). E is initialized to 2.718281828459045 and PI is initialized to 3.141592653589793. Math also declares class methods that perform various mathematics operations. Table 7-1 describes many of these methods.

Table 7-1. Math Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double abs(double d)</td>
<td>Returns the absolute value of d. There are four special cases: abs(-0.0) = +0.0, abs(+infinity) = +infinity, abs(-infinity) = +infinity, and abs(NaN) = NaN.</td>
</tr>
<tr>
<td>float abs(float f)</td>
<td>Returns the absolute value of f. There are four special cases: abs(-0.0) = +0.0, abs(+infinity) = +infinity, abs(-infinity) = +infinity, and abs(NaN) = NaN.</td>
</tr>
<tr>
<td>int abs(int i)</td>
<td>Returns the absolute value of i. There is one special case: the absolute value of Integer.MIN_VALUE is Integer.MIN_VALUE.</td>
</tr>
<tr>
<td>long abs(long l)</td>
<td>Returns the absolute value of l. There is one special case: the absolute value of Long.MIN_VALUE is Long.MIN_VALUE.</td>
</tr>
<tr>
<td>double acos(double d)</td>
<td>Returns angle d’s arc cosine within the range 0 through PI. There are three special cases: acos(anything &gt; 1) = NaN, acos(anything &lt; -1) = NaN, and acos(NaN) = NaN.</td>
</tr>
<tr>
<td>double asin(double d)</td>
<td>Returns angle d’s arc sine within the range -PI/2 through PI/2. There are three special cases: asin(anything &gt; 1) = NaN, asin(anything &lt; -1) = NaN, and asin(NaN) = NaN.</td>
</tr>
</tbody>
</table>

(continued)
### Table 7-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double atan(double d)</td>
<td>Returns angle d’s arc tangent within the range -PI/2 through PI/2. There are five special cases: $\tan(+0.0) = +0.0$, $\tan(-0.0) = -0.0$, $\tan(+\infty) = +\pi/2$, $\tan(-\infty) = -\pi/2$, and $\tan(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double ceil(double d)</td>
<td>Returns the smallest value (closest to negative infinity) that is not less than d and is equal to an integer. There are six special cases: $\text{ceil}(+0.0) = +0.0$, $\text{ceil}(-0.0) = -0.0$, $\text{ceil}(\text{anything} &gt; -1.0 \text{ and } &lt; 0.0) = -0.0$, $\text{ceil}(+\infty) = +\infty$, $\text{ceil}(-\infty) = -\infty$, and $\text{ceil}(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double cos(double d)</td>
<td>Returns the cosine of angle d (expressed in radians). There are three special cases: $\cos(\pm\infty) = \text{NaN}$, $\cos(-\infty) = \text{NaN}$, and $\cos(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double exp(double d)</td>
<td>Returns Euler’s number e to the power d. There are three special cases: $\exp(+\infty) = +\infty$, $\exp(-\infty) = +0.0$, and $\exp(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double floor(double d)</td>
<td>Returns the largest value (closest to positive infinity) that is not greater than d and is equal to an integer. There are five special cases: $\text{floor}(+0.0) = +0.0$, $\text{floor}(-0.0) = -0.0$, $\text{floor}(+\infty) = +\infty$, $\text{floor}(-\infty) = -\infty$, and $\text{floor}(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double log(double d)</td>
<td>Returns the natural logarithm (base e) of d. There are six special cases: $\log(+0.0) = -\infty$, $\log(-0.0) = -\infty$, $\log(\text{anything} &lt; 0) = \text{NaN}$, $\log(+\infty) = +\infty$, $\log(-\infty) = \text{NaN}$, and $\log(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double log10(double d)</td>
<td>Returns the base 10 logarithm of d. There are six special cases: $\log10(+0.0) = -\infty$, $\log10(-0.0) = -\infty$, $\log10(\text{anything} &lt; 0) = \text{NaN}$, $\log10(+\infty) = +\infty$, $\log10(-\infty) = \text{NaN}$, and $\log10(\text{NaN}) = \text{NaN}$.</td>
</tr>
<tr>
<td>double max(double d1, double d2)</td>
<td>Returns the most positive (closest to positive infinity) of d1 and d2. There are four special cases: $\max(\text{NaN}, \text{anything}) = \text{NaN}$, $\max(\text{anything}, \text{NaN}) = \text{NaN}$, $\max(+0.0, -0.0) = +0.0$, and $\max(-0.0, +0.0) = +0.0$.</td>
</tr>
<tr>
<td>float max(float f1, float f2)</td>
<td>Returns the most positive (closest to positive infinity) of f1 and f2. There are four special cases: $\max(\text{NaN}, \text{anything}) = \text{NaN}$, $\max(\text{anything}, \text{NaN}) = \text{NaN}$, $\max(+0.0, -0.0) = +0.0$, and $\max(-0.0, +0.0) = +0.0$.</td>
</tr>
<tr>
<td>int max(int i1, int i2)</td>
<td>Returns the most positive (closest to positive infinity) of i1 and i2.</td>
</tr>
<tr>
<td>long max(long l1, long l2)</td>
<td>Returns the most positive (closest to positive infinity) of l1 and l2.</td>
</tr>
<tr>
<td>double min(double d1, double d2)</td>
<td>Returns the most negative (closest to negative infinity) of d1 and d2. There are four special cases: $\min(\text{NaN}, \text{anything}) = \text{NaN}$, $\min(\text{anything}, \text{NaN}) = \text{NaN}$, $\min(+0.0, -0.0) = -0.0$, and $\min(-0.0, +0.0) = -0.0$.</td>
</tr>
<tr>
<td>float min(float f1, float f2)</td>
<td>Returns the most negative (closest to negative infinity) of f1 and f2. There are four special cases: $\min(\text{NaN}, \text{anything}) = \text{NaN}$, $\min(\text{anything}, \text{NaN}) = \text{NaN}$, $\min(+0.0, -0.0) = -0.0$, and $\min(-0.0, +0.0) = -0.0$.</td>
</tr>
<tr>
<td>int min(int i1, int i2)</td>
<td>Returns the most negative (closest to negative infinity) of i1 and i2.</td>
</tr>
<tr>
<td>long min(long l1, long l2)</td>
<td>Returns the most negative (closest to negative infinity) of l1 and l2.</td>
</tr>
<tr>
<td>double random()</td>
<td>Returns a pseudorandom number between 0.0 (inclusive) and 1.0 (exclusive).</td>
</tr>
</tbody>
</table>

(continued)
Table 7-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long round(double d)</td>
<td>Returns the result of rounding d to a long integer. The result is equivalent to ((\text{long})) Math.floor((d + 0.5)). There are seven special cases: round((+0.0)) = +0.0, round((-0.0)) = +0.0, round((\text{anything} &gt; \text{Long.MAX_VALUE})) = Long.MAX_VALUE, round((\text{anything} &lt; \text{Long.MIN_VALUE})) = Long.MIN_VALUE, round((\text{+infinity})) = Long.MAX_VALUE, round((\text{-infinity})) = Long.MIN_VALUE, and round((\text{NaN})) = +0.0.</td>
</tr>
<tr>
<td>int round(float f)</td>
<td>Returns the result of rounding f to an integer. The result is equivalent to ((\text{int})) Math.floor((f + 0.5)). There are seven special cases: round((+0.0)) = +0.0, round((-0.0)) = +0.0, round((\text{anything} &gt; \text{Integer.MAX_VALUE})) = Integer.MAX_VALUE, round((\text{anything} &lt; \text{Integer.MIN_VALUE})) = Integer.MIN_VALUE, round((\text{+infinity})) = Integer.MAX_VALUE, round((\text{-infinity})) = Integer.MIN_VALUE, and round((\text{NaN})) = +0.0.</td>
</tr>
<tr>
<td>double signum(double d)</td>
<td>Returns the sign of d as -1.0 ((d &lt; 0.0)), 0.0 ((d = 0.0)), and 1.0 ((d &gt; 0.0)). There are five special cases: signum((+0.0)) = +0.0, signum((-0.0)) = -0.0, signum((\text{+infinity})) = +1.0, signum((\text{-infinity})) = -1.0, and signum((\text{NaN})) = NaN.</td>
</tr>
<tr>
<td>float signum(float f)</td>
<td>Returns the sign of f as -1.0 ((f &lt; 0.0)), 0.0 ((f = 0.0)), and 1.0 ((f &gt; 0.0)). There are five special cases: signum((+0.0)) = +0.0, signum((-0.0)) = -0.0, signum((\text{+infinity})) = +1.0, signum((\text{-infinity})) = -1.0, and signum((\text{NaN})) = NaN.</td>
</tr>
<tr>
<td>double sin(double d)</td>
<td>Returns the sine of angle d ((\text{expressed in radians})). There are five special cases: sin((+0.0)) = +0.0, sin((-0.0)) = -0.0, sin((\text{+infinity})) = NaN, sin((\text{-infinity})) = NaN, and sin((\text{NaN})) = NaN.</td>
</tr>
<tr>
<td>double sqrt(double d)</td>
<td>Returns the square root of d. There are five special cases: sqrt((+0.0)) = +0.0, sqrt((-0.0)) = -0.0, sqrt((\text{anything &lt; 0})) = NaN, sqrt((\text{+infinity})) = +infinity, and sqrt((\text{NaN})) = NaN.</td>
</tr>
<tr>
<td>double tan(double d)</td>
<td>Returns the tangent of angle d ((\text{expressed in radians})). There are five special cases: tan((+0.0)) = +0.0, tan((-0.0)) = -0.0, tan((\text{+infinity})) = NaN, tan((\text{-infinity})) = NaN, and tan((\text{NaN})) = NaN.</td>
</tr>
<tr>
<td>double toDegrees(double angrad)</td>
<td>Converts angle angrad from radians to degrees via expression angrad (*) 180 / PI. There are five special cases: toDegrees((+0.0)) = +0.0, toDegrees((-0.0)) = -0.0, toDegrees((\text{+infinity})) = +infinity, toDegrees((\text{-infinity})) = -infinity, and toDegrees((\text{NaN})) = NaN.</td>
</tr>
<tr>
<td>double toRadians(double angdeg)</td>
<td>Converts angle angdeg from degrees to radians via expression angdeg / 180 (*) PI. There are five special cases: toRadians((+0.0)) = +0.0, toRadians((-0.0)) = -0.0, toRadians((\text{+infinity})) = +infinity, toRadians((\text{-infinity})) = -infinity, and toRadians((\text{NaN})) = NaN.</td>
</tr>
</tbody>
</table>

Table 7-1 reveals a wide variety of useful math-oriented methods. For example, each abs() method returns its argument’s absolute value (number without regard for sign).

abs(double) and abs(float) are useful for comparing double precision floating-point and floating-point values safely. For example, 0.3 == 0.1 + 0.1 + 0.1 evaluates to false because 0.1 has no exact representation. However, you can compare these expressions with abs() and a tolerance.
value, which indicates an acceptable range of error. For example, Math.abs(0.3 - (0.1 + 0.1 + 0.1)) < 0.1 returns true because the absolute difference between 0.3 and 0.1 + 0.1 + 0.1 is less than a 0.1 tolerance value.

random() (which returns a number that appears to be randomly chosen but is actually chosen by a predictable math calculation and hence is pseudorandom) is useful in simulations (as well as in games and wherever an element of chance is needed). However, its double precision floating-point range of 0.0 through (almost) 1.0 isn't practical. To make random() more useful, its return value must be transformed into a more useful range, perhaps integer values 0 through 49, or maybe -100 through 100. You will find the following rnd() method useful for making these transformations:

```java
static int rnd(int limit)
{
    return (int) (Math.random() * limit);
}
```

rnd() transforms random()’s 0.0 to (almost) 1.0 double precision floating-point range to a 0 through limit - 1 integer range. For example, rnd(50) returns an integer ranging from 0 through 49. Also, -100 + rnd(201) transforms 0.0 to (almost) 1.0 into -100 through 100 by adding a suitable offset and passing an appropriate limit value.

**Caution** Don't specify (int) Math.random() * limit because this expression always evaluates to 0. The expression first casts random()’s double precision floating-point fractional value (0.0 through 0.99999...) to integer 0 by truncating the fractional part and then multiplies 0 by limit, which results in 0.

The sin() and cos() methods implement the sine and cosine trigonometric functions—see http://en.wikipedia.org/wiki/Trigonometric_functions. These functions have uses ranging from the study of triangles to modeling periodic phenomena (such as simple harmonic motion—see http://en.wikipedia.org/wiki/Simple_harmonic_motion).

You can use sin() and cos() to generate and display sine and cosine waves. Listing 7-1 presents the source code for an application that does just this.

**Listing 7-1. Graphing Sine and Cosine Waves**

```java
public class Graph
{
    final static int ROWS = 11; // Must be odd
    final static int COLS= 23;

    public static void main(String[] args)
    {
        char[][] screen = new char[ROWS][COLS];
        for (int row = 0; row < screen.length; row++)
            for (int col = 0; col < screen[0].length; col++)
                screen[row][col] = ' ';
        double scaleX = COLS / 360.0;
```
for (int degree = 0; degree < 360; degree++)
{
    int row = ROWS / 2 +
        (int) Math.round(ROWS / 2 * Math.sin(Math.toRadians(degree)));
    int col = (int) (degree * scaleX);
    screen[row][col] = 'S';
    row = ROWS / 2 +
        (int) Math.round(ROWS / 2 * Math.cos(Math.toRadians(degree)));
}

for (int row = ROWS - 1; row >= 0; row--)
{
    for (int col = 0; col < COLS; col++)
        System.out.print(screen[row][col]);
    System.out.println();
}
}

Listing 7-1 introduces a Graph class that first declares a pair of constants: ROWS and COLS. These constants specify the dimensions of an array on which the graphs are generated. ROWS must be assigned an odd integer; otherwise, an instance of the java.lang.ArrayIndexOutOfBoundsException class is thrown.

**Tip** It's a good idea to use constants wherever possible. The source code is easier to maintain because you only need to change the constant's value in one place instead of having to change each corresponding value throughout the source code.

Graph next declares its main() method, which first creates a two-dimensional screen array of characters and initializes this array to spaces. This array is used to simulate an old-style character-based screen for viewing the graphs.

main() next calculates a horizontal scale value for scaling each graph horizontally so that 360 horizontal (degree) positions fit into the number of columns specified by COLS.

Continuing, main() enters a for loop that, for each of the sine and cosine graphs, creates (row, column) coordinates for each degree value, and assigns a character to the screen array at those coordinates. The character is 'S' for the sine graph, 'C' for the cosine graph, and '*' when the cosine graph intersects the sine graph.

The row calculation invokes toRadians() to convert its degree argument to radians, which is required by the sin() and cos() methods. The value returned from sin() or cos() (-1.0 to 1.0) is then multiplied by ROWS / 2 to scale this value to half the number of rows in the screen array. After rounding the result to the nearest long integer via the long round(double d) method, a cast is used to convert from long integer to integer, and this integer is added to ROWS / 2 to offset the row coordinate so that it's relative to the array's middle row. The column calculation is simpler, multiplying the degree value by the horizontal scale factor.
The screen array is dumped to the standard output device via a pair of nested for loops. The outer for loop reverses the array output so that it appears right side up—row number 0 should output last.

Compile Listing 7-1 (javac Graph.java) and run the application (java Graph). You will observe the following output:

```
CC SSSS CC
CSSS SS CC
S*C SS CC
S CC SS CC
SS CC SS CC
S CC S CC S
C SS C SS
CC SS CC S
CC SCC SS
CC CSS SSS
CCCCC SSSS
```

Table 7-1 also reveals some curiosities beginning with +infinity, -infinity, +0.0, -0.0, and NaN (Not a Number).

Java’s floating-point calculations are capable of returning +infinity, -infinity, +0.0, -0.0, and NaN because Java largely conforms to IEEE 754 (http://en.wikipedia.org/wiki/IEEE_754), a standard for floating-point calculations. The following are the circumstances under which these special values arise:

- +infinity returns from attempting to divide a positive number by 0.0. For example, System.out.println(1.0 / 0.0); outputs Infinity.
- -infinity returns from attempting to divide a negative number by 0.0. For example, System.out.println(-1.0 / 0.0); outputs -Infinity.
- NaN returns from attempting to divide 0.0 by 0.0, attempting to calculate the square root of a negative number, and attempting other strange operations. For example, System.out.println(0.0 / 0.0); and System.out.println(Math.sqrt(-1.0)); each output NaN.
- +0.0 results from attempting to divide a positive number by +infinity. For example, System.out.println(1.0 / (1.0 / 0.0)); outputs 0.0 (+0.0 without the + sign).
- -0.0 results from attempting to divide a negative number by +infinity. For example, System.out.println(-1.0 / (1.0 / 0.0)); outputs -0.0.

After an operation yields +infinity, -infinity, or NaN, the rest of the expression usually equals that special value. For example, System.out.println(1.0 / 0.0 * 20.0); outputs Infinity. Also, an expression that first yields +infinity or -infinity might devolve into NaN. For example, expression 1.0 / 0.0 * 0.0 first yields +infinity (1.0 / 0.0) and then yields NaN (+infinity * 0.0).
Another curiosity is Integer.MAX_VALUE, Integer.MIN_VALUE, Long.MAX_VALUE, and Long.MIN_VALUE. Each of these constants identifies the maximum or minimum value that can be represented by the class’s associated primitive type. (You’ll learn more about these classes later in this chapter.)

Finally, you might wonder why the abs(), max(), and min() overloaded methods don’t include byte and short versions, as in byte abs(byte b) and short abs(short s). There is no need for these methods because the limited ranges of bytes and short integers make them unsuitable in calculations. If you need such a method, check out Listing 7-2.

**Listing 7-2. Obtaining Absolute Values for Byte Integers and Short Integers**

```java
public class AbsByteShort {
    static byte abs(byte b) {
        return (b < 0) ? (byte) -b : b;
    }

    static short abs(short s) {
        return (s < 0) ? (short) -s : s;
    }

    public static void main(String[] args) {
        byte b = -2;
        System.out.println(abs(b)); // Output: 2
        short s = -3;
        System.out.println(abs(s)); // Output: 3
    }
}
```

Listing 7-2’s (byte) and (short) casts are necessary because -b converts b’s value from a byte to an int, and -s converts s’s value from a short to an int. In contrast, these casts are not needed with (b < 0) and (s < 0), which automatically cast b’s and s’s values to an int before comparing them with int-based 0.

**Note**  Their absence from Math suggests that byte and short are not very useful in method declarations. However, these types are useful when declaring arrays whose elements store small values (such as a binary file’s byte values). If you declared an array of int or long to store such values, you would end up wasting heap space (and might even run out of memory).
StrictMath and strictfp

While searching through the java.lang package documentation, you will probably encounter a class named StrictMath. Apart from a longer name, this class appears to be identical to Math. The differences between these classes can be summed up as follows:

- StrictMath’s methods return exactly the same results on all platforms. In contrast, some of Math’s methods might return values that vary ever so slightly from platform to platform.
- Because StrictMath cannot utilize platform-specific features such as an extended-precision math coprocessor, an implementation of StrictMath might be less efficient than an implementation of Math.

For the most part, Math’s methods call their StrictMath counterparts. Two exceptions are toDegrees() and toRadians(). Although these methods have identical code bodies in both classes, StrictMath’s implementations include reserved word strictfp in the method headers:

```java
public static strictfp double toDegrees(double angrad)
public static strictfp double toRadians(double angdeg)
```

Wikipedia’s “strictfp” entry (http://en.wikipedia.org/wiki/Strictfp) mentions that strictfp restricts floating-point calculations to ensure portability. This reserved word accomplishes portability in the context of intermediate floating-point representations and overflows/underflows (generating a value too large or small to fit a representation).

Without strictfp, an intermediate calculation is not limited to the IEEE 754 32-bit and 64-bit floating-point representations that Java supports. Instead, the calculation can take advantage of a larger representation (perhaps 128 bits) on a platform that supports this representation.

An intermediate calculation that overflows or underflows when its value is represented in 32/64 bits might not overflow/underflow when its value is represented in more bits. Because of this discrepancy, portability is compromised. strictfp levels the playing field by requiring all platforms to use 32/64 bits for intermediate calculations.

When applied to a method, strictfp ensures that all floating-point calculations performed in that method are in strict compliance. However, strictfp can be used in a class header declaration (as in public strictfp class FourierTransform) to ensure that all floating-point calculations performed in that class are strict.

**Note**  Math and StrictMath are declared final so that they cannot be extended. Also, they declare private empty noargument constructors so that they cannot be instantiated. Finally, Math and StrictMath are examples of utility classes because they exist as placeholders for utility constants and utility (static) methods.
Exploring Number and Its Children

The abstract java.lang.Number class is the superclass of those classes representing numeric values that are convertible to the byte integer, double precision floating-point, floating-point, integer, long integer, and short integer primitive types. This class offers the following conversion methods:

- byte byteValue()
- double doubleValue()
- float floatValue()
- int intValue()
- long longValue()
- short shortValue()

You typically don’t work with Number directly, unless you’ve created a collection or array of Number subclass objects and plan to iterate over this collection/array, calling one of the conversion methods on each stored instance. Instead, you would typically work with one of the following subclasses:

- java.util.concurrent.atomic.AtomicInteger
- java.util.concurrent.atomic.AtomicLong
- java.math.BigDecimal
- java.math.BigInteger
- java.lang.Byte
- java.lang.Double
- java.lang.Float
- java.lang.Integer
- java.lang.Long
- java.lang.Short

I’ll shortly discuss all of these types except for AtomicInteger and AtomicLong, which I’ll discuss in Chapter 10.

BigDecimal

In Chapter 3, I introduced a SavingsAccount class with a balance field of type int. This field records the number of dollars in this account. Alternatively, it could represent the number of pennies that the account contains.
Perhaps you are wondering why I didn’t declare \texttt{balance} to be of type \texttt{double} or \texttt{float}. That way, \texttt{balance} could store values such as 18.26 (18 dollars in the whole number part and 26 pennies in the fraction part). I didn’t declare \texttt{balance} to be a \texttt{double} or \texttt{float} for the following reasons:

- Not all floating-point values that can represent monetary amounts (dollars and cents) can be stored exactly in memory. For example, 0.1 (which you might use to represent 10 cents), has no exact storage representation. If you executed

\begin{verbatim}
double total = 0.1; for (int i = 0; i < 50; i++) total += 0.1; System.out.println(total);
\end{verbatim}

you would observe 5.099999999999998 instead of the correct 5.1 as the output.

- The result of each floating-point calculation needs to be rounded to the nearest cent. Failure to do so introduces tiny errors that can cause the final result to differ from the correct result. Although \texttt{Math} supplies a pair of \texttt{round()} methods that you might consider using to round a calculation to the nearest cent, these methods round to the nearest integer (dollar).

Listing 7-3’s \texttt{InvoiceCalc} application demonstrates both problems. However, the first problem isn’t serious because it contributes very little to the inaccuracy. The more serious problem occurs from failing to round to the nearest cent after performing a calculation.

\textbf{Listing 7-3. Floating-Point-Based Invoice Calculations Leading to Confusing Results}

\begin{verbatim}
import java.text.NumberFormat;

public class InvoiceCalc
{
    final static double DISCOUNT_PERCENT = 0.1; // 10%
    final static double TAX_PERCENT = 0.05; // 5%

    public static void main(String[] args)
    {
        double invoiceSubtotal = 285.36;
        double discount = invoiceSubtotal * DISCOUNT_PERCENT;
        double subtotalBeforeTax = invoiceSubtotal - discount;
        double salesTax = subtotalBeforeTax * TAX_PERCENT;
        double invoiceTotal = subtotalBeforeTax + salesTax;
        NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
        System.out.println("Subtotal: " + currencyFormat.format(invoiceSubtotal));
        System.out.println("Discount: " + currencyFormat.format(discount));
        System.out.println("SubTotal after discount: " +
             currencyFormat.format(subtotalBeforeTax));
        System.out.println("Sales Tax: " + currencyFormat.format(salesTax));
        System.out.println("Total: " + currencyFormat.format(invoiceTotal));
    }
}
\end{verbatim}

Listing 7-3 performs several invoice-related calculations that result in an incorrect final total. After performing these calculations, it obtains a currency-based formatter for formatting double precision floating-point values into string-based monetary amounts with a currency symbol (such as the dollar sign [$]). The formatter is obtained by calling the \texttt{java.text.NumberFormat} class’s \texttt{NumberFormat}
getCurrencyInstance() method (see Chapter 16). A value is then formatted into a currency string by passing this value as an argument to NumberFormat's String format(double value) method.

When you run InvoiceCalc, you will discover the following output:

Subtotal: $285.36
Discount: $28.54
SubTotal after discount: $256.82
Sales Tax: $12.84
Total: $269.67

This output reveals the correct subtotal, discount, subtotal after discount, and sales tax. In contrast, it incorrectly gives 269.67 instead of 269.66 as the final total. The customer will probably not appreciate paying an extra penny, even though 269.67 is the correct value according to the floating-point calculations:

Subtotal: 285.36
Discount: 28.536
SubTotal after discount: 256.824
Sales Tax: 12.8412
Total: 269.6652

The problem arises from not rounding the result of each calculation to the nearest cent before performing the next calculation. As a result, the 0.024 in 256.824 and 0.0012 in 12.84 contribute to the final value, causing NumberFormat's format() method to round this value to 269.67.

Caution Never use float or double to represent monetary values.

Java provides a solution to both problems in the form of a BigDecimal class. This immutable class (a BigDecimal instance cannot be modified) represents a signed decimal number (such as 23.653) of arbitrary precision (number of digits) with an associated scale (an integer that specifies the number of digits after the decimal point).

BigDecimal declares three convenience constants: ONE, TEN, and ZERO. Each constant is the BigDecimal equivalent of 1, 10, and 0 with a zero scale.

Caution BigDecimal declares several ROUND_-_prefixed constants. These constants are largely obsolete and should be avoided, along with the public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) and public BigDecimal setScale(int newScale, int roundingMode) methods, which are still present so that dependent legacy code continues to compile.

BigDecimal also declares a variety of useful constructors and methods. A few of these constructors and methods are described in Table 7-2.
Table 7-2. BigDecimal Constructors and Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal(int val)</td>
<td>Initializes the BigDecimal instance to val's digits. Sets the scale to 0.</td>
</tr>
<tr>
<td>BigDecimal(String val)</td>
<td>Initializes the BigDecimal instance to the decimal equivalent of val. Sets the scale to the number of digits after the decimal point or 0 when no decimal point is specified. This constructor throws java.lang.NullPointerException when val is null and java.lang.NumberFormatException when val's string representation is invalid (contains letters, for example).</td>
</tr>
<tr>
<td>BigDecimal abs()</td>
<td>Returns a new BigDecimal instance that contains the absolute value of the current instance's value. The resulting scale is the same as the current instance's scale.</td>
</tr>
<tr>
<td>BigDecimal add(BigDecimal augend)</td>
<td>Returns a new BigDecimal instance that contains the sum of the current value and the argument value. The resulting scale is the maximum of the current and argument scales. This method throws NullPointerException when augend is null.</td>
</tr>
<tr>
<td>BigDecimal divide(BigDecimal divisor)</td>
<td>Returns a new BigDecimal instance that contains the quotient of the current value divided by the argument value. The resulting scale is the difference of the current and argument scales. It might be adjusted when the result requires more digits. This method throws NullPointerException when divisor is null or java.lang.ArithmeticException when divisor represents 0 or the result cannot be represented exactly.</td>
</tr>
<tr>
<td>BigDecimal max(BigDecimal val)</td>
<td>Returns either this or val, whichever BigDecimal instance contains the larger value. This method throws NullPointerException when val is null.</td>
</tr>
<tr>
<td>BigDecimal min(BigDecimal val)</td>
<td>Returns either this or val, whichever BigDecimal instance contains the smaller value. This method throws NullPointerException when val is null.</td>
</tr>
<tr>
<td>BigDecimal multiply(BigDecimal multiplicand)</td>
<td>Returns a new BigDecimal instance that contains the product of the current value and the argument value. The resulting scale is the sum of the current and argument scales. This method throws NullPointerException when multiplicand is null.</td>
</tr>
<tr>
<td>BigDecimal negate()</td>
<td>Returns a new BigDecimal instance that contains the negative of the current value. The resulting scale is the same as the current scale.</td>
</tr>
<tr>
<td>int precision()</td>
<td>Returns the precision of the current BigDecimal instance.</td>
</tr>
<tr>
<td>BigDecimal remainder(BigDecimal divisor)</td>
<td>Returns a new BigDecimal instance that contains the remainder of the current value divided by the argument value. The resulting scale is the difference of the current scale and the argument scale. It might be adjusted when the result requires more digits. This method throws NullPointerException when divisor is null or ArithmeticException when divisor represents 0.</td>
</tr>
<tr>
<td>int scale()</td>
<td>Returns the scale of the current BigDecimal instance.</td>
</tr>
</tbody>
</table>
Table 7-2. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal setScale(int newScale, RoundingMode roundingMode)</td>
<td>Returns a new BigDecimal instance with the specified scale and rounding mode. If the new scale is greater than the old scale, additional zeros are added to the unscaled value. In this case, no rounding is necessary. If the new scale is smaller than the old scale, trailing digits are removed. If these trailing digits are not zero, the remaining unscaled value has to be rounded. For this rounding operation, the specified rounding mode is used. This method throws NullPointerException when roundingMode is null, and ArithmeticException when roundingMode is set to RoundingMode.ROUND_UNNECESSARY but rounding is necessary based on the current scale.</td>
</tr>
<tr>
<td>BigDecimal subtract(BigDecimal subtrahend)</td>
<td>Returns a new BigDecimal instance that contains the current value minus the argument value. The resulting scale is the maximum of the current and argument scales. This method throws NullPointerException when subtrahend is null.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string representation of this BigDecimal instance. Scientific notation is used when necessary.</td>
</tr>
</tbody>
</table>

Table 7-2 refers to java.math.RoundingMode, which is an enum containing various rounding mode constants. These constants are described in Table 7-3.

Table 7-3. RoundingMode Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEILING</td>
<td>Rounds toward positive infinity.</td>
</tr>
<tr>
<td>DOWN</td>
<td>Rounds toward zero.</td>
</tr>
<tr>
<td>FLOOR</td>
<td>Rounds toward negative infinity.</td>
</tr>
<tr>
<td>HALF_DOWN</td>
<td>Rounds toward the “nearest neighbor” unless both neighbors are equidistant, in which case rounds down.</td>
</tr>
<tr>
<td>HALF_EVEN</td>
<td>Rounds toward the “nearest neighbor” unless both neighbors are equidistant, in which case rounds toward the even neighbor.</td>
</tr>
<tr>
<td>HALF_UP</td>
<td>Rounds toward the “nearest neighbor” unless both neighbors are equidistant, in which case rounds up. (This is the rounding mode commonly taught at school.)</td>
</tr>
<tr>
<td>UNNECESSARY</td>
<td>Rounding isn’t necessary because the requested operation produces the exact result.</td>
</tr>
<tr>
<td>UP</td>
<td>Positive values are rounded toward positive infinity and negative values are rounded toward negative infinity.</td>
</tr>
</tbody>
</table>

The best way to get comfortable with BigDecimal is to try it out. Listing 7-4 uses this class to correctly perform the invoice calculations that were presented in Listing 7-3.
Listing 7-4. BigDecimal-Based Invoice Calculations Not Leading to Confusing Results

```java
import java.math.BigDecimal;
import java.math.RoundingMode;

public class InvoiceCalc
{
    public static void main(String[] args)
    {
        BigDecimal invoiceSubtotal = new BigDecimal("285.36");
        BigDecimal discountPercent = new BigDecimal("0.10");
        BigDecimal discount = invoiceSubtotal.multiply(discountPercent);
        discount = discount.setScale(2, RoundingMode.HALF_UP);
        BigDecimal subtotalBeforeTax = invoiceSubtotal.subtract(discount);
        subtotalBeforeTax = subtotalBeforeTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal salesTaxPercent = new BigDecimal("0.05");
        BigDecimal salesTax = subtotalBeforeTax.multiply(salesTaxPercent);
        salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal invoiceTotal = subtotalBeforeTax.add(salesTax);
        invoiceTotal = invoiceTotal.setScale(2, RoundingMode.HALF_UP);
        System.out.println("Subtotal: " + invoiceSubtotal);
        System.out.println("Discount: " + discount);
        System.out.println("SubTotal after discount: " + subtotalBeforeTax);
        System.out.println("Sales Tax: " + salesTax);
        System.out.println("Total: " + invoiceTotal);
    }
}
```

Listing 7-4’s main() method first creates BigDecimal objects invoiceSubtotal and discountPercent that are initialized to 285.36 and 0.10, respectively. It multiplies invoiceSubtotal by discountPercent and assigns the BigDecimal result to discount.

At this point, discount contains 28.5360. Apart from the trailing zero, this value is the same as that generated by invoiceSubtotal * DISCOUNT_PERCENT in Listing 7-3. The value that should be stored in discount is 28.54. To correct this problem before performing another calculation, main() calls discount’s setScale() method with these arguments:

- 2: Two digits after the decimal point
- RoundingMode.HALF_UP: The conventional approach to rounding

After setting the scale and proper rounding mode, main() subtracts discount from invoiceSubtotal and assigns the resulting BigDecimal instance to subtotalBeforeTax. main() calls setScale() on subtotalBeforeTax to properly round its value before moving on to the next calculation.

main() next creates a BigDecimal object named salesTaxPercent that is initialized to 0.05. It then multiplies subtotalBeforeTax by salesTaxPercent, assigning the result to salesTax, and calls setScale() on this BigDecimal object to properly round its value.

Moving on, main() adds salesTax to subtotalBeforeTax, saving the result in invoiceTotal, and rounds the result via setScale(). The values in these objects are sent to the standard output device via System.out.println(), which calls their toString() methods to return string representations of the BigDecimal values.
When you run this new version of InvoiceCalc, you will discover the following output:

Subtotal: 285.36
Discount: 28.54
SubTotal after discount: 256.82
Sales Tax: 12.84
Total: 269.66

Caution  BigDecimal declares a BigDecimal(double val) constructor that you should avoid using if at all possible. This constructor initializes the BigDecimal instance to the value stored in val, making it possible for this instance to reflect an invalid representation when the double cannot be stored exactly. For example, BigDecimal(0.1) results in 0.1000000000000000055511151231257827021181583404541015625 being stored in the instance. In contrast, BigDecimal("0.1") stores 0.1 exactly.

BigInteger

BigDecimal stores a signed decimal number as an unscaled value with a 32-bit integer scale. The unscaled value is stored in an instance of the BigInteger class.

BigInteger is an immutable class that represents a signed integer of arbitrary precision. It stores its value in two’s complement format (all bits are flipped—1s to 0s and 0s to 1s—and 1 is added to the result to be compatible with the two’s complement format used by Java’s byte integer, short integer, integer, and long integer types).

Note  Check out Wikipedia’s “Two’s complement” entry (http://en.wikipedia.org/wiki/Two%27s_complement) to learn more about two’s complement.

BigInteger declares three convenience constants: ONE, TEN, and ZERO. Each constant is the BigInteger equivalent of 1, 10, and 0.

BigInteger also declares a variety of useful constructors and methods. A few of these constructors and methods are described in Table 7-4.
### Table 7-4. BigInteger Constructors and Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigInteger(byte[] val)</td>
<td>Initializes the BigInteger instance to the integer that is stored in the val array, with val[0] storing the integer’s most significant (leftmost) 8 bits. This constructor throws NullPointerException when val is null and NumberFormatException when val.length equals 0.</td>
</tr>
<tr>
<td>BigInteger(String val)</td>
<td>Initializes the BigInteger instance to the integer equivalent of val. This constructor throws NullPointerException when val is null and NumberFormatException when val’s string representation is invalid (contains letters, for example).</td>
</tr>
<tr>
<td>BigInteger abs()</td>
<td>Returns a new BigInteger instance that contains the absolute value of the current instance’s value.</td>
</tr>
<tr>
<td>BigInteger add(BigInteger augend)</td>
<td>Returns a new BigInteger instance that contains the sum of the current value and the argument value. This method throws NullPointerException when augend is null.</td>
</tr>
<tr>
<td>BigInteger divide(BigInteger divisor)</td>
<td>Returns a new BigInteger instance that contains the quotient of the current value divided by the argument value. This method throws NullPointerException when divisor is null and ArithmeticException when divisor represents 0 or the result cannot be represented exactly.</td>
</tr>
<tr>
<td>BigInteger max(BigInteger val)</td>
<td>Returns either this or val, whichever BigInteger instance contains the larger value. This method throws NullPointerException when val is null.</td>
</tr>
<tr>
<td>BigInteger min(BigInteger val)</td>
<td>Returns either this or val, whichever BigInteger instance contains the smaller value. This method throws NullPointerException when val is null.</td>
</tr>
<tr>
<td>BigInteger multiply(BigInteger multiplicand)</td>
<td>Returns a new BigInteger instance that contains the product of the current value and the argument value. This method throws NullPointerException when multiplicand is null.</td>
</tr>
<tr>
<td>BigInteger negate()</td>
<td>Returns a new BigInteger instance that contains the negative of the current value.</td>
</tr>
<tr>
<td>BigInteger remainder(BigInteger divisor)</td>
<td>Returns a new BigInteger instance that contains the remainder of the current value divided by the argument value. This method throws NullPointerException when divisor is null and ArithmeticException when divisor represents 0.</td>
</tr>
<tr>
<td>BigInteger subtract(BigInteger subtrahend)</td>
<td>Returns a new BigInteger instance that contains the current value minus the argument value. This method throws NullPointerException when subtrahend is null.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string representation of this BigInteger instance.</td>
</tr>
</tbody>
</table>
Note BigInteger also declares several bit-oriented methods, such as BigInteger and(BigInteger val), BigInteger flipBit(int n), and BigInteger shiftLeft(int n). These methods are useful for when you need to perform low-level bit manipulation.

The best way to get comfortable with BigInteger is to try it out. Listing 7-5 uses this class in a factorial() method comparison context.

Listing 7-5. Comparing factorial() Methods

```java
import java.math.BigInteger;

public class FactComp {
    public static void main(String[] args) {
        System.out.println(factorial(12));
        System.out.println();
        System.out.println(factorial(20L));
        System.out.println();
        System.out.println(factorial(170.0));
        System.out.println();
        System.out.println(factorial(new BigInteger("170")));
        System.out.println();
        System.out.println(factorial(new BigInteger("25")));
    }

    static int factorial(int n) {
        if (n == 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    static long factorial(long n) {
        if (n == 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    static double factorial(double n) {
        if (n == 1.0)
            return 1.0;
        else
            return n * factorial(n - 1);
    }
```
else
    return n * factorial(n - 1);
}

static BigInteger factorial(BigInteger n)
{
    if (n.equals(BigInteger.ZERO))
        return BigInteger.ONE;
    else
        return n.multiply(factorial(n.subtract(BigInteger.ONE)));
}

Listing 7-5 compares four versions of the recursive factorial() method. This comparison reveals the largest argument that can be passed to each of the first three methods before the returned factorial value becomes meaningless because of limits on the range of values that can be accurately represented by the numeric type.

The first version is based on int and has a useful argument range of 0 through 12. Passing any argument greater than 12 results in a factorial that cannot be represented accurately as an int.

You can increase the useful range of factorial(), but not by much, by changing the parameter and return types to long. After making these changes, you will discover that the upper limit of the useful range is 20.

To further increase the useful range, you might create a version of factorial() whose parameter and return types are double. This is possible because whole numbers can be represented exactly as doubles. However, the largest useful argument that can be passed is 170.0. Anything higher than this value results in factorial() returning +infinity.

It’s possible that you might need to calculate a higher factorial value, perhaps in the context of calculating a statistics problem involving combinations or permutations. The only way to accurately calculate this value is to use a version of factorial() based on BigInteger.

When you run the previous application, it generates the following output:

479001600
2432902008176640000
7.257415615307994E306
72574156153079989673967282111292631147169916812964513765435779890056184340170615785235074924261745
951149099123783852077666602256544275302532890077320751090240043028005829560396661259965825710439855
8294257568966313439612262571094946806711205568880457193340212661452800000000000000000000000000000000000
0000000000
1.5511210043330986E25
15511210043330985984000000
The first three values represent the highest factorials that can be returned by the int-based, long-based, and double-based factorial() methods. The fourth value represents the BigInteger equivalent of the highest double factorial.

Notice that the double method fails to accurately represent 170! (! is the math symbol for factorial). Its precision is simply too small. Although the method attempts to round the smallest digit, rounding doesn’t always work—the number ends in 7994 instead of 7998. Rounding is only accurate up to argument 25.0, as the last two output lines reveal.

Note RSA encryption (http://en.wikipedia.org/wiki/RSA_(algorithm)) offers another use for BigInteger.

**Primitive Type Wrapper Classes**

Byte, Double, Float, Integer, Long, and Short along with Boolean and Character are known as primitive type wrapper classes or value classes because their instances wrap themselves around values of primitive types. Java provides these eight primitive type wrapper classes for two reasons:

- The Collections Framework (discussed in Chapter 9) provides lists, sets, and maps that can only store objects; they cannot store primitive-type values. You store a primitive-type value in a primitive type wrapper class instance and store the instance in the collection.

- These classes provide a good place to associate useful constants (such as MAX_VALUE and MIN_VALUE) and class methods (such as Integer’s parseInt() methods and Character’s isDigit(), isLetter(), and toUpperCase() methods) with the primitive types.

In this section, I will introduce you to each of these primitive type wrapper classes.

**Boolean**

Boolean is the smallest of the primitive type wrapper classes. This class declares three constants, including TRUE and FALSE, which denote precreated Boolean objects. It also declares a pair of constructors for initializing a Boolean object:

- Boolean(boolean value) initializes the Boolean object to value.
- Boolean(String s) converts s’s text to a true or false value and stores this value in the Boolean object.

The second constructor compares s’s value with true. Because the comparison is case insensitive, any uppercase/lowercase combination of these four letters (such as true, TRUE, or tRue) results in true being stored in the object. Otherwise, the constructor stores false in the object.
Boolean also declares or overrides the following methods:

- `int compareTo(Boolean b)` compares the current Boolean object with `b` to determine their relative order. The method returns 0 when the current object contains the same Boolean value as `b`, a positive value when the current object contains true and `b` contains false, and a negative value when the current object contains false and `b` contains true.

- `boolean equals(Object o)` compares the current Boolean object with `o` and returns true when `o` is not null, `o` is of type Boolean, and both objects contain the same Boolean value.

- `static boolean getBoolean(String name)` returns true when a system property (discussed later in this chapter) identified by `name` exists and is equal to true.

- `int hashCode()` returns a suitable hash code that allows Boolean objects to be used with hash-based collections (discussed in Chapter 9).

- `static boolean parseBoolean(String s)` parses `s`, returning true when `s` equals "true", "TRUE", "True", or any other uppercase/lowercase combination. Otherwise, this method returns false. (*Parsing* breaks a sequence of characters into meaningful components, known as *tokens*.)

- `String toString()` returns "true" when the current Boolean instance contains true; otherwise, this method returns "false".

- `static String toString(boolean b)` returns "true" when `b` contains true; otherwise, this method returns "false".

- `static Boolean valueOf(boolean b)` returns TRUE when `b` contains true or FALSE when `b` contains false.

- `static Boolean valueOf(String s)` returns TRUE when `s` equals "true", "TRUE", "True", or any other uppercase/lowercase combination. Otherwise, this method returns FALSE.

**Note** Boolean's constructors are complemented by `boolean booleanValue()`, which returns the wrapped Boolean value.

**Caution** Newcomers to the Boolean class often think that `getBoolean()` returns a Boolean object's true/false value. However, `getBoolean()` returns the value of a Boolean-based system property. I will discuss system properties later in this chapter. If you need to return a Boolean object's true/false value, use the `booleanValue()` method instead.
It's often better to use `TRUE` and `FALSE` than to create `Boolean` objects. For example, consider a method that returns a `Boolean` object containing `true` when the method's `double` argument is negative or `false` when this argument is zero or positive. You might declare your method like the following `isNegative()` method:

```java
Boolean isNegative(double d)
{
   return new Boolean(d < 0);
}
```

Although this method is concise, it unnecessarily creates a `Boolean` object. When the method is called frequently, many `Boolean` objects are created that consume heap space. When heap space runs low, the garbage collector runs and slows down the application, which impacts performance.

The following example reveals a better way to code `isNegative()`:

```java
Boolean isNegative(double d)
{
   return (d < 0) ? Boolean.TRUE : Boolean.FALSE;
}
```

This method avoids creating `Boolean` objects by returning either the precreated `TRUE` or `FALSE` object.

**Tip** You should strive to create as few objects as possible. Not only will your applications have smaller memory footprints, they'll perform better because the garbage collector will not run as often.

**Character**

`Character` is the largest of the primitive type wrapper classes, containing many constants, a constructor, many methods, and a pair of nested classes (`Subset` and `UnicodeBlock`).

**Note** `Character`'s complexity derives from Java's support for Unicode (http://en.wikipedia.org/wiki/Unicode). For brevity, I ignore much of `Character`'s Unicode-related complexity, which is beyond the scope of this chapter.

`Character` declares a single `Character(char value)` constructor, which you use to initialize a `Character` object to `value`. This constructor is complemented by `char charValue()`, which returns the wrapped character value.
When you start writing applications, you might codify expressions such as \( \text{ch} \geq '0' \&\& \text{ch} \leq '9' \) (test \text{ch} to see if it contains a digit) and \( \text{ch} \geq 'A' \&\& \text{ch} \leq 'Z' \) (test \text{ch} to see if it contains an uppercase letter). You should avoid doing so for three reasons:

- It's too easy to introduce a bug into the expression. For example, \( \text{ch} > '0' \&\& \text{ch} \leq '9' \) introduces a subtle bug that doesn't include '0' in the comparison.
- The expressions are not very descriptive of what they are testing.
- The expressions are biased toward Latin digits (0–9) and letters (A–Z and a–z). They don't take into account digits and letters that are valid in other languages. For example, '௫' is a character literal representing one of the digits in the Tamil language.

Character declares several comparison and conversion class methods that address these concerns. These methods include the following:

- `static boolean isDigit(char ch)` returns true when \( \text{ch} \) contains a digit (typically 0 through 9 but also digits in other alphabets).
- `static boolean isLetter(char ch)` returns true when \( \text{ch} \) contains a letter (typically A–Z or a–z but also letters in other alphabets).
- `static boolean isLetterOrDigit(char ch)` returns true when \( \text{ch} \) contains a letter or digit (typically A–Z, a–z, or 0–9 but also letters or digits in other alphabets).
- `static boolean isLowerCase(char ch)` returns true when \( \text{ch} \) contains a lowercase letter.
- `static boolean isUpperCase(char ch)` returns true when \( \text{ch} \) contains an uppercase letter.
- `static boolean isWhitespace(char ch)` returns true when \( \text{ch} \) contains a whitespace character (typically a space, a horizontal tab, a carriage return, or a line feed).
- `static char toLowerCase(char ch)` returns the lowercase equivalent of \( \text{ch} \)’s uppercase letter; otherwise, this method returns \( \text{ch} \)’s value.
- `static char toUpperCase(char ch)` returns the uppercase equivalent of \( \text{ch} \)’s lowercase letter; otherwise, this method returns \( \text{ch} \)’s value.

For example, `isDigit(ch)` is preferable to \( \text{ch} \geq '0' \&\& \text{ch} \leq '9' \) because it avoids a source of bugs, is more readable, and returns true for non-Latin digits (such as '௫') and Latin digits.

**Float and Double**

Float and Double store floating-point and double precision floating-point values in Float and Double objects, respectively. These classes declare the following constants:

- `MAX_VALUE` identifies the maximum value that can be represented as a float or double.
- `MIN_VALUE` identifies the minimum value that can be represented as a float or double.
**NaN** represents \(0.0F / 0.0F\) as a float and \(0.0 / 0.0\) as a double.

**NEGATIVE_INFINITY** represents \(-\infty\) as a float or double.

**POSITIVE_INFINITY** represents \(+\infty\) as a float or double.

Float and Double also declare the following constructors for initializing their objects:

- Float(float value) initializes the Float object to value.
- Float(double value) initializes the Float object to the float equivalent of value.
- Float(String s) converts s's text to a floating-point value and stores this value in the Float object.
- Double(double value) initializes the Double object to value.
- Double(String s) converts s's text to a double precision floating-point value and stores this value in the Double object.

Float's constructors are complemented by float floatValue(), which returns the wrapped floating-point value. Similarly, Double's constructors are complemented by double doubleValue(), which returns the wrapped double precision floating-point value.

Float declares several utility methods in addition to floatValue(). These methods include the following:

- static int floatToIntBits(float value) converts value to a 32-bit integer.
- static boolean isInfinite(float f) returns true when f's value is +infinity or -infinity. A related boolean isInfinite() method returns true when the current Float object's value is +infinity or -infinity.
- static boolean isNaN(float f) returns true when f's value is NaN. A related boolean isNaN() method returns true when the current Float object's value is NaN.
- static float parseFloat(String s) parses s, returning the floating-point equivalent of s's textual representation of a floating-point value or throwing NumberFormatException when this representation is invalid (contains letters, for example).

Double declares several utility methods as well as doubleValue(). These methods include the following:

- static long doubleToLongBits(double value) converts value to a long integer.
- static boolean isInfinite(double d) returns true when d's value is +infinity or -infinity. A related boolean isInfinite() method returns true when the current Double object's value is +infinity or -infinity.
- static boolean isNaN(double d) returns true when d's value is NaN. A related public boolean isNaN() method returns true when the current Double object's value is NaN.
- static double parseDouble(String s) parses s, returning the double precision floating-point equivalent of s's textual representation of a double precision floating-point value or throwing NumberFormatException when this representation is invalid.
The `floatToIntBits()` and `doubleToIntBits()` methods are used in implementations of the `equals()` and `hashCode()` methods that must take `float` and `double` fields into account. `floatToIntBits()` and `doubleToIntBits()` allow `equals()` and `hashCode()` to respond properly to the following situations:

- `equals()` must return true when `f1` and `f2` contain `Float.NaN` (or `d1` and `d2` contain `Double.NaN`). If `equals()` was implemented in a manner similar to `f1.floatValue() == f2.floatValue()` (or `d1.doubleValue() == d2.doubleValue()`), this method would return false because NaN is not equal to anything, including itself.

- `equals()` must return false when `f1` contains +0.0 and `f2` contains -0.0 (or vice versa), or `d1` contains +0.0 and `d2` contains -0.0 (or vice versa). If `equals()` was implemented in a manner similar to `f1.floatValue() == f2.floatValue()` (or `d1.doubleValue() == d2.doubleValue()`), this method would return true because +0.0 == -0.0 returns true.

These requirements are needed for hash-based collections (discussed in Chapter 9) to work properly. Listing 7-6 shows how they impact `Float`'s and `Double`'s `equals()` methods.

**Listing 7-6. Demonstrating `Float`'s `equals()` Method in a NaN Context and `Double`'s `equals()` Method in a +/-0.0 Context**

```java
public class FloatDoubleDemo {
    public static void main(String[] args) {
        Float f1 = new Float(Float.NaN);
        System.out.println(f1.floatValue());
        Float f2 = new Float(Float.NaN);
        System.out.println(f2.floatValue());
        System.out.println(f1.equals(f2));
        System.out.println(Float.NaN == Float.NaN);
        System.out.println();
        Double d1 = new Double(+0.0);
        System.out.println(d1.doubleValue());
        Double d2 = new Double(-0.0);
        System.out.println(d2.doubleValue());
        System.out.println(d1.equals(d2));
        System.out.println(+0.0 == -0.0);
    }
}
```

Compile Listing 7-6 (javac FloatDoubleDemo.java) and run this application (java FloatDoubleDemo). The following output proves that `Float`'s `equals()` method properly handles NaN and `Double`'s `equals()` method properly handles +/-0.0:

NaN
NaN
true
false
When you want to test a float or double value for equality with +infinity or -infinity (but not both), don't use `isInfinite()`. Instead, compare the value with `NEGATIVE_INFINITY` or `POSITIVE_INFINITY` via `==`. For example, `f == Float.NEGATIVE_INFINITY`.

You will find `parseFloat()` and `parseDouble()` useful in many contexts. For example, Listing 7-7 uses `parseDouble()` to parse command-line arguments into doubles.

**Listing 7-7. Parsing Command-Line Arguments into Double Precision Floating-Point Values**

```java
public class Calc {
    public static void main(String[] args) {
        if (args.length != 3) {
            System.err.println("usage: java Calc value1 op value2");
            System.err.println("op is one of +, -, x, or /");
            return;
        }
        try {
            double value1 = Double.parseDouble(args[0]);
            double value2 = Double.parseDouble(args[2]);
            if (args[1].equals("+"))
                System.out.println(value1 + value2);
            else if (args[1].equals("-"))
                System.out.println(value1 - value2);
            else if (args[1].equals("x"))
                System.out.println(value1 * value2);
            else if (args[1].equals(="/"))
                System.out.println(value1 / value2);
            else
                System.err.println("invalid operator: " + args[1]);
        }
    }
}
```
catch (NumberFormatException nfe)
{
    System.err.println("Bad number format: " + nfe.getMessage());
}
}

Specify java Calc 10E+3 + 66.0 to try out the Calc application. This application responds by outputting 10066.0. If you specified java Calc 10E+3 + A instead, you would observe Bad number format: For input string: "A" as the output, which is in response to the second parseDouble() method call's throwing of a NumberFormatException object.

Although NumberFormatException describes an unchecked exception, and although unchecked exceptions are often not handled because they represent coding mistakes, NumberFormatException doesn't fit this pattern in this example. The exception doesn't arise from a coding mistake; it arises from someone passing an illegal numeric argument to the application, which cannot be avoided through proper coding. Perhaps NumberFormatException should have been implemented as a checked exception.

### Integer, Long, Short, and Byte

Integer, Long, Short, and Byte store 32-bit, 64-bit, 16-bit, and 8-bit integer values in Integer, Long, Short, and Byte objects, respectively.

Each class declares MAX_VALUE and MIN_VALUE constants that identify the maximum and minimum values that can be represented by its associated primitive type. These classes also declare the following constructors for initializing their objects:

- Integer(int value) initializes the Integer object to value.
- Integer(String s) converts s's text to a 32-bit integer value and stores this value in the Integer object.
- Long(long value) initializes the Long object to value.
- Long(String s) converts s's text to a 64-bit integer value and stores this value in the Long object.
- Short(short value) initializes the Short object to value.
- Short(String s) converts s's text to a 16-bit integer value and stores this value in the Short object.
- Byte(byte value) initializes the Byte object to value.
- Byte(String s) converts s's text to an 8-bit integer value and stores this value in the Byte object.

Integer's constructors are complemented by int intValue(), Long's constructors are complemented by long longValue(), Short's constructors are complemented by short shortValue(), and Byte's constructors are complemented by byte byteValue(). These inherited methods return wrapped integers.
These classes declare various useful integer-oriented methods. For example, Integer declares the following class methods for converting a 32-bit integer to a String according to a specific representation (binary, hexadecimal, octal, and decimal):

- `static String toBinaryString(int i)` returns a String object containing i’s binary representation. For example, `Integer.toBinaryString(255)` returns a String object containing 11111111.
- `static String toHexString(int i)` returns a String object containing i’s hexadecimal representation. For example, `Integer.toHexString(255)` returns a String object containing ff.
- `static String toOctalString(int i)` returns a String object containing i’s octal representation. For example, `toOctalString(64)` returns a String object containing 100.
- `static String toString(int i)` returns a String object containing i’s decimal representation. For example, `toString(255)` returns a String object containing 255.

It’s often convenient to prepend zeros to a binary string so that you can align multiple binary strings in columns. For example, you might want to create an application that displays the following aligned output:

```
11110001
+ 00000111
--------
11111000
```

Unfortunately, `toBinaryString()` doesn’t let you accomplish this task. For example, `Integer.toBinaryString(7)` returns a String object containing 111 instead of 00000111. Listing 7-8’s `toAlignedBinaryString()` method addresses this oversight.

**Listing 7-8. Aligning Binary Strings**

```java
public class AlignBinaryString {

    public static void main(String[] args) {
        System.out.println(toAlignedBinaryString(7, 8));
        System.out.println(toAlignedBinaryString(255, 16));
        System.out.println(toAlignedBinaryString(255, 7));
    }

    static String toAlignedBinaryString(int i, int numBits) {
        String result = Integer.toBinaryString(i);
        if (result.length() > numBits)
            return null; // cannot fit result into numBits columns
        int numLeadingZeros = numBits - result.length();
        StringBuilder sb = new StringBuilder();
        for (int j = 0; j < numLeadingZeros; j++)
            sb.append('0');
        return sb.append(result).toString();
    }
}
```

www.it-ebooks.info
```java
sb.append('0');
return sb.toString() + result;
}
```

The `toAlignedBinaryString()` method takes two arguments: the first argument specifies the 32-bit integer that is to be converted into a binary string, and the second argument specifies the number of bit columns in which to fit the string.

After calling `toBinaryString()` to return `i`'s equivalent binary string without leading zeros, `toAlignedBinaryString()` verifies that the string's digits can fit into the number of bit columns specified by `numBits`. If they don't fit, this method returns null.

Moving on, `toAlignedBinaryString()` calculates the number of leading "0"s to prepend to `result` and then uses a `for` loop to create a string of leading zeros. This method ends by returning the leading zeros string prepended to the result string.

When you run this application, it generates the following output:

```
00000111
0000000011111111
null
```

### Exploring String, StringBuffer, and StringBuilder

Many computer languages implement the concept of a **string**, a sequence of characters treated as a single unit (and not as individual characters). For example, the C language implements a string as an array of characters terminated by the null character (`'\0'`). In contrast, Java implements a string via the `java.lang.String` class.

String objects are immutable: you cannot modify a `String` object's string. The various `String` methods that appear to modify the `String` object actually return a new `String` object with modified string content instead. Because returning new `String` objects is often wasteful, Java provides the `java.lang.StringBuffer` and equivalent `java.lang.StringBuilder` classes as a workaround.

This section introduces you to `String`, `StringBuilder`, and `StringBuffer`.

### String

String represents a string as a sequence of characters. In contrast to strings in the C language, this sequence is not terminated by a null character. Instead, its length is stored separately.

You typically obtain a `String` by assigning a string literal to a variable of `String` type; for example, `String favLanguage = "Java";`. You can also obtain a `String` by calling a `String` constructor; for example, `String favLanguage = new String("Java");`.

After obtaining a `String`, you can invoke methods to accomplish various tasks. For example, you can obtain a string's length by invoking the `length()` method; for example, `favLanguage.length()`.

Table 7-5 describes some of `String`'s constructors and methods for initializing `String` objects and working with strings.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String(char[] data)</td>
<td>Initializes this String object to the characters in the data array. Modifying data after initializing this String object has no effect on the object.</td>
</tr>
<tr>
<td>String(String s)</td>
<td>Initializes this String object to s's string.</td>
</tr>
<tr>
<td>char charAt(int index)</td>
<td>Returns the character located at the zero-based index in this String object's string. Throws java.lang.StringIndexOutOfBoundsException when index is less than 0 or greater than or equal to the length of the string.</td>
</tr>
<tr>
<td>String concat(String s)</td>
<td>Returns a new String object containing this String object's string followed by the s argument's string.</td>
</tr>
<tr>
<td>boolean endsWith(String suffix)</td>
<td>Returns true when this String object's string ends with the characters in the suffix argument, when suffix is empty (contains no characters), or when suffix contains the same character sequence as this String object's string. This method performs a case-sensitive comparison (a is not equal to A, for example) and throws NullPointerException when suffix is null.</td>
</tr>
<tr>
<td>boolean equals(Object object)</td>
<td>Returns true when object is of type String and this argument's string contains the same characters (and in the same order) as this String object's string.</td>
</tr>
<tr>
<td>boolean equalsIgnoreCase(String s)</td>
<td>Returns true when s and this String object contain the same characters (ignoring case). This method returns false when the character sequences differ or when null is passed to s.</td>
</tr>
<tr>
<td>int indexOf(int c)</td>
<td>Returns the zero-based index of the first occurrence (from the start of the string to the end of the string) of the character represented by c in this String object's string. Returns -1 when this character is not present.</td>
</tr>
<tr>
<td>int indexOf(String s)</td>
<td>Returns the zero-based index of the first occurrence (from the start of the string to the end of the string) of s's character sequence in this String object's string. Returns -1 when s is not present. Throws NullPointerException when s is null.</td>
</tr>
<tr>
<td>String intern()</td>
<td>Searches an internal table of String objects for an object whose string is equal to this String object's string. This String object's string is added to the table when not present. Returns the object contained in the table whose string is equal to this String object's string. The same String object is always returned for strings that are equal.</td>
</tr>
<tr>
<td>int lastIndexOf(int c)</td>
<td>Returns the zero-based index of the last occurrence (from the start of the string to the end of the string) of the character represented by c in this String object's string. Returns -1 when this character is not present.</td>
</tr>
<tr>
<td>int lastIndexOf(String s)</td>
<td>Returns the zero-based index of the last occurrence (from the start of the string to the end of the string) of s's character sequence in this String object's string. Returns -1 when s is not present. Throws NullPointerException when s is null.</td>
</tr>
<tr>
<td>int length()</td>
<td>Returns the number of characters in this String object's string.</td>
</tr>
</tbody>
</table>

(continued)
Table 7-5. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String replace(char oldChar, char newChar)</td>
<td>Returns a new String object whose string matches this String object’s string except that all occurrences of oldChar have been replaced by newChar.</td>
</tr>
<tr>
<td>String[] split(String expr)</td>
<td>Splits this String object’s string into an array of String objects using the regular expression (a string whose pattern [template] is used to search a string for substrings that match the pattern) specified by expr as the basis for the split. Throws NullPointerException when expr is null and java.util.regex.PatternSyntaxException when expr’s syntax is invalid.</td>
</tr>
<tr>
<td>boolean startsWith(String prefix)</td>
<td>Returns true when this String object’s string starts with the characters in the prefix argument, when prefix is empty (contains no characters), or when prefix contains the same character sequence as this String object’s string. This method performs a case-sensitive comparison (such as a is not equal to A), and throws NullPointerException when prefix is null.</td>
</tr>
<tr>
<td>String substring(int start)</td>
<td>Returns a new String object whose string contains this String object’s characters beginning with the character located at start. Throws StringIndexOutOfBoundsException when start is negative or greater than the length of this String object’s string.</td>
</tr>
<tr>
<td>char[] toCharArray()</td>
<td>Returns a character array that contains the characters in this String object’s string.</td>
</tr>
<tr>
<td>String toLowerCase()</td>
<td>Returns a new String object whose string contains this String object’s characters where uppercase letters have been converted to lowercase. This String object is returned when it contains no uppercase letters to convert.</td>
</tr>
<tr>
<td>String toUpperCase()</td>
<td>Returns a new String object whose string contains this String object’s characters where lowercase letters have been converted to uppercase. This String object is returned when it contains no lowercase letters to convert.</td>
</tr>
<tr>
<td>String trim()</td>
<td>Returns a new String object that contains this String object’s string with whitespace characters (characters whose Unicode values are 32 or less) removed from the start and end of the string, or this String object when there is no leading/trailing whitespace.</td>
</tr>
</tbody>
</table>

Table 7-5 reveals a couple of interesting items about String. First, this class’s String(String s) constructor doesn’t initialize a String object to a string literal, as in new String("Java"). Instead, it behaves similarly to the C++ copy constructor by initializing the String object to the contents of another String object. This behavior suggests that a string literal is more than it appears to be.

In reality, a string literal is a String object. You can prove this to yourself by executing System.out.println("abc".length()); and System.out.println("abc" instanceof String);. The first method call outputs 3, which is the length of the "abc" String object’s string, and the second method call outputs true ("abc" is a String object).
CHAPTER 7: Exploring the Basic APIs, Part 1

**Note**  String literals are stored in a classfile data structure known as the *constant pool*. When a class is loaded, a String object is created for each literal and is stored in an internal table of String objects.

The second interesting item is the `intern()` method, which *interns* (stores a unique copy of) a String object in an internal table of String objects. `intern()` makes it possible to compare strings via their references and `==` or `!=`. These operators are the fastest way to compare strings, which is especially valuable when sorting a huge number of strings.

Listing 7-9 demonstrates these concepts.

**Listing 7-9. Demonstrating a Couple of Interesting Things About Strings**

```java
public class StringDemo {
    public static void main(String[] args) {
        System.out.println("abc".length());
        System.out.println("abc" instanceof String);
        System.out.println("abc" == "a" + "bc");
        System.out.println("abc" == new String("abc").intern());
    }
}
```

Compile Listing 7-9 (`javac StringDemo.java`) and run this application (`java StringDemo`). You should observe the following output:

```
3
true
true
false
true
```

By default, String objects denoted by literal strings ("abc") and string-valued constant expressions ("a" + "bc") are interned, which is why `System.out.println("abc" == "a" + "bc");` outputs true. However, String objects created via String constructors are not interned, which is why `System.out.println("abc" == new String("abc").intern());` outputs false. In contrast, `System.out.println("abc" == new String("abc").intern());` outputs true.

**Caution**  Be careful with this string comparison technique (which only compares references) because you can easily introduce a bug when one of the strings being compared has not been interned. When in doubt, use the `equals()` or `equalsIgnoreCase()` method. For example, each of "abc".equals(new String("abc")) and "abc".equalsIgnoreCase(new String("ABC")) returns true.
Table 7-5 also reveals the `charAt()` method, which is useful for extracting a string’s characters. Listing 7-10 offers a demonstration.

**Listing 7-10. Iterating Over a String**

```java
public class StringDemo {
    public static void main(String[] args) {
        String s = "abc";
        for (int i = 0; i < s.length(); i++)
            System.out.println(s.charAt(i));
    }
}
```

Compile Listing 7-10 and run this application. You will observe that for `(int i = 0; i < s.length(); i++) System.out.println(s.charAt(i));` returns each of `s`’s `a`, `b`, and `c` characters and outputs it on a separate line.

Finally, Table 7-5 presents `split()`, a method that I employed in Chapter 6’s `StubFinder` application to split a string’s comma-separated list of values into an array of `String` objects. This method uses a regular expression that identifies a sequence of characters around which the string is split. (I will discuss regular expressions in Chapter 13.)

---

**Note**  
`StringIndexOutOfBoundsException` and `ArrayIndexOutOfBoundsException` are sibling classes that share a common `java.lang.IndexOutOfBoundsException` superclass.

---

**StringBuffer and StringBuilder**

String objects are immutable: you cannot modify a String object’s string. The `String` methods that appear to modify the `String` object (such as `replace()`) actually return a new `String` object with modified string content instead. Because returning new `String` objects is often wasteful, Java provides the `StringBuffer` and `StringBuilder` classes as a workaround.

`StringBuffer` and `StringBuilder` are identical apart from the fact that `StringBuilder` offers better performance than `StringBuffer` but cannot be used in the context of multiple threads without explicit synchronization. (I discuss threads and synchronization later in this chapter.)

---

**Tip**  
Use `StringBuffer` in a multithreaded context (for safety) and `StringBuilder` in a single-threaded context (for performance).
StringBuffer and StringBuilder provide an internal character array for building a string efficiently. After creating a StringBuffer/StringBuilder object, you call various methods to append, delete, and insert the character representations of various values to, from, and into the array. You then call toString() to convert the array’s content to a String object and return this object.

Table 7-6 describes some of StringBuffer’s constructors and methods for initializing StringBuffer objects and working with string buffers. StringBuilder’s constructors and methods are identical and won't be discussed.

**Table 7-6. StringBuffer Constructors and Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringBuffer()</td>
<td>Initializes this StringBuffer object to an empty array with an initial capacity of 16 characters.</td>
</tr>
<tr>
<td>StringBuffer(int capacity)</td>
<td>Initializes this StringBuffer object to an empty array with an initial capacity of capacity characters. This constructor throws java.lang.NegativeArraySizeException when capacity is negative.</td>
</tr>
<tr>
<td>StringBuffer(String s)</td>
<td>Initializes this StringBuffer object to an array containing s’s characters. This object’s initial capacity is 16 plus the length of s. This constructor throws NullPointerException when s is null.</td>
</tr>
<tr>
<td>StringBuffer append(boolean b)</td>
<td>Appends “true” to this StringBuffer object’s array when b is true and “false” to the array when b is false, and returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(char ch)</td>
<td>Appends ch’s character to this StringBuffer object’s array, and returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(char[] chars)</td>
<td>Appends the characters in the chars array to this StringBuffer object’s array, and returns this StringBuffer object. This method throws NullPointerException when chars is null.</td>
</tr>
<tr>
<td>StringBuffer append(double d)</td>
<td>Appends the string representation of d’s double precision floating-point value to this StringBuffer object’s array, and returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(float f)</td>
<td>Appends the string representation of f’s floating-point value to this StringBuffer object’s array, and returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(int i)</td>
<td>Appends the string representation of i’s integer value to this StringBuffer object’s array, and returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(long l)</td>
<td>Appends the string representation of l’s long integer value to this StringBuffer object’s array, and returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(Object obj)</td>
<td>Calls obj’s toString() method and appends the returned string’s characters to this StringBuffer object’s array. Appends “null” to the array when null is passed to obj. Returns this StringBuffer object.</td>
</tr>
<tr>
<td>StringBuffer append(String s)</td>
<td>Appends s’s string to this StringBuffer object’s array. Appends “null” to the array when null is passed to s. Returns this StringBuffer object.</td>
</tr>
<tr>
<td>int capacity()</td>
<td>Returns the current capacity of this StringBuffer object’s array.</td>
</tr>
</tbody>
</table>

(continued)
### Table 7-6. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char charAt(int index)</td>
<td>Returns the character located at index in this StringBuffer object's array. This method throws StringIndexOutOfBoundsException when index is negative or greater than or equal to this StringBuffer object's length.</td>
</tr>
<tr>
<td>void ensureCapacity(int min)</td>
<td>Ensures that this StringBuffer object's capacity is at least that specified by min. If the current capacity is less than min, a new internal array is created with greater capacity. The new capacity is set to the larger of min and the current capacity multiplied by 2, with 2 added to the result. No action is taken when min is negative or zero.</td>
</tr>
<tr>
<td>int length()</td>
<td>Returns the number of characters stored in this StringBuffer object's array.</td>
</tr>
<tr>
<td>StringBuffer reverse()</td>
<td>Returns this StringBuffer object with its array contents reversed.</td>
</tr>
<tr>
<td>void setCharAt(int index, char ch)</td>
<td>Replaces the character at index with ch. This method throws StringIndexOutOfBoundsException when index is negative or greater than or equal to the length of this StringBuffer object's array.</td>
</tr>
<tr>
<td>void setLength(int length)</td>
<td>Sets the length of this StringBuffer object's array to length. If the length argument is less than the current length, the array's contents are truncated. If the length argument is greater than or equal to the current length, sufficient null characters (\u0000) are appended to the array. This method throws StringIndexOutOfBoundsException when length is negative.</td>
</tr>
<tr>
<td>String substring(int start)</td>
<td>Returns a new String object that contains all characters in this StringBuffer object's array starting with the character located at start. This method throws StringIndexOutOfBoundsException when start is less than 0 or greater than or equal to the length of this StringBuffer object's array.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a new String object whose string equals the contents of this StringBuffer object's array.</td>
</tr>
</tbody>
</table>

A StringBuffer or StringBuilder object’s internal array is associated with the concepts of capacity and length. *Capacity* refers to the maximum number of characters that can be stored in the array before the array grows to accommodate additional characters. *Length* refers to the number of characters that are already stored in the array.

Listing 7-11 demonstrates StringBuffer(), various append() methods, and toString().

**Listing 7-11. Demonstrating StringBuffer**

```java
public class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello,");
        sb.append(' ');
        sb.append("world. ");
        sb.append(args.length);
        sb.append(" argument(s) have been passed to this method.");
    }
}
```

www.it-ebooks.info
string s = sb.toString();
System.out.println(s);
}
}

After creating a StringBuffer object initialized to the contents of the interned "Hello," String object, main() appends a character, another String object, an integer (identifying the number of command-line arguments passed to this application), and yet another String object to this buffer. It then converts the StringBuffer's contents to a String object and prints this object's content on the standard output device.

Compile Listing 7-11 (javac StringBufferDemo.java) and run this application (java StringBufferDemo). You should observe the following output:

Hello, world. 0 argument(s) have been passed to this method.

Re-run this application with one or more command-line arguments and the number will change to reflect how many command-line arguments were passed.

Consider a scenario where you've written code to format an integer value into a string. As part of the formatter, you need to prepend a specific number of leading spaces to the integer. You decide to use the following initialization code and loop to build a spacesPrefix string with three leading spaces:

```java
int numLeadingSpaces = 3; // default value
String spacesPrefix = "";
for (int j = 0; j < numLeadingSpaces; j++)
    spacesPrefix += "0";
```

This loop is inefficient because each of the iterations creates a StringBuilder object and a String object. The compiler transforms this code fragment into the following fragment:

```java
int numLeadingSpaces = 3; // default value
String spacesPrefix = "";
for (int j = 0; j < numLeadingSpaces; j++)
    spacesPrefix = new StringBuilder().append(spacesPrefix).append("0").toString();
```

A more efficient way to code the previous loop involves creating a StringBuilder/StringBuffer object prior to entering the loop, calling the appropriate append() method in the loop, and calling toString() after the loop. The following code fragment demonstrates this more efficient scenario:

```java
int numLeadingSpaces = 3; // default value
StringBuilder sb = new StringBuilder();
for (int j = 0; j < numLeadingSpaces; j++)
    sb.append('0');
String spacesPrefix = sb.toString();
```

**Caution** Avoid using the string concatenation operator in a lengthy loop because it results in the creation of many unnecessary StringBuilder and String objects.
Exploring System

The java.lang.System utility class declares class methods that provide access to the current time (in milliseconds), system property values, environment variable values, and other kinds of system information. Furthermore, it declares class methods that support the system tasks of copying one array to another array, requesting garbage collection, and so on.

Table 7-7 describes some of System’s methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)</td>
<td>Copies the number of elements specified by length from the src array starting at zero-based offset srcPos into the dest array starting at zero-based offset destPos. This method throws NullPointerException when src or dest is null, IndexOutOfBoundsException when copying causes access to data outside array bounds, and java.lang.ArrayStoreException when an element in the src array cannot be stored into the dest array because of a type mismatch.</td>
</tr>
<tr>
<td>long currentTimeMillis()</td>
<td>Returns the current system time in milliseconds since January 1, 1970 00:00:00 UTC (Coordinated Universal Time—see <a href="http://en.wikipedia.org/wiki/Coordinated_Universal_Time">http://en.wikipedia.org/wiki/Coordinated_Universal_Time</a>).</td>
</tr>
<tr>
<td>void gc()</td>
<td>Informs the virtual machine that now would be a good time to run the garbage collector. This is only a hint; there is no guarantee that the garbage collector will run.</td>
</tr>
<tr>
<td>String getEnv(String name)</td>
<td>Returns the value of the environment variable identified by name.</td>
</tr>
<tr>
<td>String getProperty(String name)</td>
<td>Returns the value of the system property (platform-specific attribute, such as a version number) identified by name, or returns null when such a property doesn’t exist. Examples of system properties that are useful in an Android context include file.separator, java.class.path, java.home, java.io.tmpdir, java.library.path, line.separator, os.arch, os.name, path.separator, and user.dir.</td>
</tr>
<tr>
<td>void runFinalization()</td>
<td>Informs the virtual machine that now would be a good time to perform any outstanding object finalizations. This is only a hint; there is no guarantee that outstanding object finalizations will be performed.</td>
</tr>
</tbody>
</table>

**Note** System declares SecurityManager getSecurityManager() and void setSecurityManager(SecurityManager sm) methods that are not supported by Android. On an Android device, the former method always returns null, and the latter method always throws an instance of the java.lang.SecurityException class. Regarding the latter method, its documentation states that “security managers do not provide a secure environment for executing untrusted code and are unsupported on Android. Untrusted code cannot be safely isolated within a single virtual machine on Android.”

www.it-ebooks.info
Listing 7-12 demonstrates the arraycopy(), currentTimeMillis(), and getProperty() methods.

Listing 7-12. Experimenting with System Methods

```java
public class SystemDemo {
    public static void main(String[] args) {
        int[] grades = { 86, 92, 78, 65, 43, 72, 98, 81 }; 
        int[] gradesBackup = new int[grades.length];
        System.arraycopy(grades, 0, gradesBackup, 0, grades.length);
        for (int i = 0; i < gradesBackup.length; i++)
            System.out.println(gradesBackup[i]);
        System.out.println("Current time: " + System.currentTimeMillis());
        String[] propNames = {
            "file.separator",
            "java.class.path",
            "java.home",
            "java.io.tmpdir",
            "java.library.path",
            "line.separator",
            "os.arch",
            "os.name",
            "path.separator",
            "user.dir"
        };
        for (int i = 0; i < propNames.length; i++)
            System.out.println(propNames[i] + ": " + System.getProperty(propNames[i]));
    }
}
```

Listing 7-12’s `main()` method begins by demonstrating `arraycopy()`. It uses this method to copy the contents of a grades array to a gradesBackup array.

**Tip** The `arraycopy()` method is the fastest portable way to copy one array to another. Also, when you write a class whose methods return a reference to an internal array, you should use `arraycopy()` to create a copy of the array and then return the copy’s reference. That way, you prevent clients from directly manipulating (and possibly screwing up) the internal array.

`main()` next calls `currentTimeMillis()` to return the current time as a milliseconds value. Because this value is not human-readable, you might want to use the `java.util.Date` class (discussed in Chapter 16). The `Date()` constructor calls `currentTimeMillis()` and its `toString()` method converts this value to a readable date and time.

`main()` concludes by demonstrating `getProperty()` in a for loop. This loop iterates over all of Table 7-7’s property names, outputting each name and value.
Compile Listing 7-12 (javac SystemDemo.java) and run this application (java SystemDemo). When I run this application on my platform, it generates the following output:

86
92
78
65
52
43
72
98
91
Current time: 1353115138889
file.separator: \
java.class.path: .;C:\Program Files (x86)\QuickTime\QTSystem\QTJava.zip
java.home: C:\Program Files\Java\jre7
java.io.tmpdir: C:\Users\Owner\AppData\Local\Temp\njava.library.path: C:\Windows\system32;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:\ Program Files (x86)\AMD APP\bin\x86_64;C:\Program Files (x86)\AMD APP\bin\x86;C:\Program Files\ Common Files\Microsoft Shared\Windows Live;C:\Program Files (x86)\Common Files\Microsoft Shared\ Windows Live;C:\Windows\system32;C:\Windows;C:\Windows\system32\Wbem;C:\Windows\System32\ WindowsPowerShell\v1.0;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static;C:\Program Files (x86)\Windows Live\Shared;C:\Program Files\java\jdk1.7.0_06\bin;C:\Program Files (x86)\Borland\ BCC55\bin;C:\android;C:\android\tools;C:\android\platform-tools;C:\Program Files (x86)\apache- ant-1.8.2\bin;C:\Program Files (x86)\QuickTime\QTSystem;.
line.separator:
os.arch: amd64
os.name: Windows 7
path.separator: ;
user.dir: C:\prj\dev\ljfad2\ch08\code\SystemDemo

Note  line.separator stores the actual line separator character/characters, not its/their representation (such as \r\n), which is why a blank line appears after line.separator:

Exploring Threads

Applications execute via threads, which are independent paths of execution through an application’s code. When multiple threads are executing, each thread’s path can differ from other thread paths. For example, a thread might execute one of a switch statement’s cases, and another thread might execute another of this statement’s cases.
CHAPTER 7: Exploring the Basic APIs, Part 1

The virtual machine gives each thread its own method-call stack to prevent threads from interfering with each other. Separate stacks let threads keep track of their next instructions to execute, which can differ from thread to thread. The stack also provides a thread with its own copy of method parameters, local variables, and return value.

Java supports threads via its Threads API. This API largely consists of one interface (Runnable) and four classes (Thread, ThreadGroup, ThreadLocal, and InheritableThreadLocal) in the java.lang package. After exploring Runnable and Thread (and mentioning ThreadGroup during this exploration), I explore synchronization, ThreadLocal, and InheritableThreadLocal.

**Note** Applications use threads to improve performance. Some applications can get by with only the default main thread (the thread that executes the main() method) to carry out their tasks, but other applications need additional threads to perform time-intensive tasks in the background, so that they remain responsive to their users.

Java supports threads via its Threads API. This API largely consists of one interface (Runnable) and four classes (Thread, ThreadGroup, ThreadLocal, and InheritableThreadLocal) in the java.lang package. After exploring Runnable and Thread (and mentioning ThreadGroup during this exploration), I explore synchronization, ThreadLocal, and InheritableThreadLocal.

**Note** Java 5 introduced the java.util.concurrent package as a high-level alternative to the low-level Threads API. (I will discuss this package in Chapter 10.) Although java.util.concurrent is the preferred API for working with threads, you should also be somewhat familiar with Threads because it's helpful in simple threading scenarios. Also, you might have to analyze someone else's source code that depends on Threads.

**Runnable and Thread**

Java's Runnable interface identifies those objects that supply code for threads to execute via this interface's solitary void run() method—a thread receives no arguments and returns no value. In the following code fragment, an anonymous class implements Runnable:

```java
Runnable r = new Runnable() {
    @Override
    public void run() {
        // perform some work
    }
};
```

Java's Thread class provides a consistent interface to the underlying operating system's threading architecture. (The operating system is typically responsible for creating and managing threads.) A single operating system thread is associated with a Thread object.
Thread declares several constructors for initializing Thread objects. Some of these constructors take Runnable arguments. For example, `Thread(Runnable runnable)` initializes a new Thread object to the specified runnable whose code is to be executed, which the following code fragment demonstrates:

```java
Thread t = new Thread(r);
```

Other constructors don't take Runnable arguments. For example, `Thread()` doesn't initialize Thread to a Runnable argument. You must extend Thread and override its `run()` method to supply the code to run, which the following code fragment accomplishes:

```java
class MyThread extends Thread
{
    @Override
    public void run()
    {
    }
}
```

In the absence of an explicit name argument, each constructor assigns a unique default name (starting with Thread-) to the Thread object. Names make it possible to differentiate threads. In contrast to the previous two constructors, which choose default names, `Thread(String threadName)` lets you specify your own thread name.

Thread also declares methods for starting and managing threads. Table 7-8 describes many of the more useful methods.

**Table 7-8. Thread Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Thread currentThread()</td>
<td>Returns the Thread object associated with the thread that calls this method.</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns the name associated with this Thread object.</td>
</tr>
<tr>
<td>Thread.State getState()</td>
<td>Returns the state of the thread associated with this Thread object. The state is identified by the Thread.State enum as one of BLOCKED (waiting to acquire a lock, discussed later), NEW (created but not started), Runnable (executing), TERMINATED (the thread has died), TIMED_WAITING (waiting for a specified amount of time to elapse), or WAITING (waiting indefinitely).</td>
</tr>
<tr>
<td>void interrupt()</td>
<td>Sets the interrupt status flag in this Thread object. If the associated thread is blocked or is waiting, clear this flag and wake up the thread by throwing an instance of the java.lang.InterruptedException class.</td>
</tr>
<tr>
<td>static boolean interrupted()</td>
<td>Returns true when the thread associated with this Thread object has a pending interrupt request. Clears the interrupt status flag.</td>
</tr>
<tr>
<td>boolean isAlive()</td>
<td>Returns true to indicate that this Thread object's associated thread is alive and not dead. A thread's life span ranges from just before it is actually started within the start() method to just after it leaves the run() method, at which point it dies.</td>
</tr>
</tbody>
</table>

(continued)
Table 7-8. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isDaemon()</td>
<td>Returns true when the thread associated with this Thread object is a <em>daemon thread</em>, a thread that acts as a helper to a <em>user thread</em> (nondaemon thread) and dies automatically when the application’s last nondaemon thread dies so the application can exit.</td>
</tr>
<tr>
<td>boolean isInterrupted()</td>
<td>Returns true when the thread associated with this Thread object has a pending interrupt request.</td>
</tr>
<tr>
<td>void join()</td>
<td>The thread that calls this method on this Thread object waits for the thread associated with this object to die. This method throws InterruptedException when this Thread object's interrupt() method is called.</td>
</tr>
<tr>
<td>void join(long millis)</td>
<td>The thread that calls this method on this Thread object waits for the thread associated with this object to die, or until millis milliseconds have elapsed, whichever happens first. This method throws InterruptedException when this Thread object's interrupt() method is called.</td>
</tr>
<tr>
<td>void setDaemon(boolean isDaemon)</td>
<td>Marks this Thread object's associated thread as a daemon thread when isDaemon is true. This method throws java.lang.IllegalThreadStateException when the thread has not yet been created and started.</td>
</tr>
<tr>
<td>void setName(String threadName)</td>
<td>Assigns threadName's value to this Thread object as the name of its associated thread.</td>
</tr>
<tr>
<td>static void sleep(long time)</td>
<td>Pauses the thread associated with this Thread object for time milliseconds. This method throws InterruptedException when this Thread object's interrupt() method is called while the thread is sleeping.</td>
</tr>
<tr>
<td>void start()</td>
<td>Creates and starts this Thread object's associated thread. This method throws IllegalStateException when the thread was previously started and is running or has died.</td>
</tr>
</tbody>
</table>

Listing 7-13 introduces you to the Threads API via a main() method that demonstrates Runnable, Thread(Runnable runnable), currentThread(), getName(), and start().

Listing 7-13. A Pair of Counting Threads

```java
public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
```
while (true)
    System.out.println(name + ": " + count++);
};
Thread thdA = new Thread(r);
Thread thdB = new Thread(r);
    thdA.start();
    thdB.start();
}

According to Listing 7-13, the default main thread that executes main() first instantiates an anonymous class that implements Runnable. It then creates two Thread objects, initializing each object to the runnable, and calls Thread's start() method to create and start both threads. After completing these tasks, the main thread exits main() and dies.

Each of the two started threads executes the runnable’s run() method. It calls Thread's currentThread() method to obtain its associated Thread instance, uses this instance to call Thread's getName() method to return its name, initializes count to 0, and enters an infinite loop where it outputs name and count, and increments count on each iteration.

**Tip**  To stop an application that doesn’t end, press the Ctrl and C keys simultaneously on a Windows platform or do the equivalent on a non-Windows platform.

I observed both threads alternating in their execution when I ran this application on the 64-bit Windows 7 platform. Partial output from one run appears here:

Thread-0: 0
Thread-0: 1
Thread-1: 0
Thread-0: 2
Thread-1: 1
Thread-0: 3
Thread-1: 2
Thread-0: 4
Thread-1: 3
Thread-0: 5
Thread-1: 4
Thread-0: 6
Thread-1: 5
Thread-0: 7
Thread-1: 6
Thread-1: 7
Thread-1: 8
Thread-1: 9
Thread-1: 10
Thread-1: 11
Thread-1: 12
CHAPTER 7: Exploring the Basic APIs, Part 1

Note: I executed `java CountThreads >output.txt` to capture the output to `output.txt` and then presented part of this file’s content above. Capturing output to a file may significantly affect the output that would otherwise be observed if output wasn’t captured. Because I present captured thread output throughout this section, bear this in mind when executing the application on your platform. Also, note that your platform’s threading architecture may impact the observable results. I’ve tested each thread example on the 64-bit Windows 7 platform.

Also, although the output shows that the first thread (Thread-0) starts executing, never assume that the thread associated with the `Thread` object whose `start()` method is called first will execute first.

When a computer has enough processors and/or processor cores, the computer’s operating system assigns a separate thread to each processor or core so the threads execute simultaneously. When a computer doesn’t have enough processors and/or cores, various threads must wait their turns to use the shared processors/cores.

The operating system uses a scheduler (http://en.wikipedia.org/wiki/Scheduling_(computing)) to determine when a waiting thread executes. The following list identifies three different schedulers:

- Linux 2.6 through 2.6.22 uses the O(1) Scheduler (http://en.wikipedia.org/wiki/O(1)_scheduler).
- Windows NT-based operating systems (NT, XP, Vista, and 7) use a multilevel feedback queue scheduler (http://en.wikipedia.org/wiki/Multilevel_feedback_queue). This scheduler has been adjusted in Windows Vista and Windows 7 to optimize performance.

A multilevel feedback queue and many other thread schedulers take priority (thread relative importance) into account. They often combine preemptive scheduling (higher priority threads preempt—interrupt and run instead of—lower priority threads) with round robin scheduling (equal priority threads are given equal slices of time, which are known as time slices, and take turns executing).

Note: Two terms that are commonly encountered when exploring threads are parallelism and concurrency. According to Oracle’s “Multithreading Guide” (http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html), parallelism is “a condition that arises when at least two threads are executing simultaneously.” In contrast, concurrency is “a condition that exists when at least two threads are making progress. [It is a] more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.”
CHAPTER 7: Exploring the Basic APIs, Part 1

Caution Using the `setPriority()` method can impact an application’s portability across platforms because different schedulers can handle a priority change in different ways. For example, one platform’s scheduler might delay lower priority threads from executing until higher priority threads finish. This delaying can lead to indefinite postponement or starvation because lower priority threads “starve” while waiting indefinitely for their turn to execute, and this can seriously hurt the application’s performance. Another platform’s scheduler might not indefinitely delay lower priority threads, improving application performance.

Listing 7-14 refactors Listing 7-13’s `main()` method to give each thread a nondefault name and to put each thread to sleep after outputting name and count.

Listing 7-14. A Pair of Counting Threads Revisited

```java
public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true) {
                    System.out.println(name + " : " + count++);
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ie) {
                    }
                }
            }
        };
        Thread thdA = new Thread(r);
        thdA.setName("A");
        Thread thdB = new Thread(r);
        thdB.setName("B");
    }
}
```
Listing 7-14 reveals that threads A and B execute `Thread.sleep(100);` to sleep for 100 milliseconds. This sleep results in each thread executing more frequently, as the following partial output reveals:

```
A: 0
B: 0
A: 1
B: 1
B: 2
A: 2
B: 3
A: 3
B: 4
A: 4
B: 5
A: 5
B: 6
A: 6
B: 7
A: 7
```

A thread will occasionally start another thread to perform a lengthy calculation, download a large file, or perform some other time-consuming activity. After finishing its other tasks, the thread that started the **worker thread** is ready to process the results of the worker thread and waits for the worker thread to finish and die.

It's possible to wait for the worker thread to die by using a while loop that repeatedly calls `Thread.isAlive()` method on the worker thread's `Thread` object and sleeps for a certain length of time when this method returns true. However, Listing 7-15 demonstrates a less verbose alternative: the `join()` method.

**Listing 7-15. Joining the Default Main Thread with a Background Thread**

```java
class JoinDemo {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Worker thread is simulating work by sleeping for 5 seconds.");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        new Thread(r, "Worker Thread").start();
        new Thread(r, "Worker Thread").start();
    }
}
```

```java
public class JoinDemo {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Worker thread is simulating work by sleeping for 5 seconds.");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        new Thread(r, "Worker Thread").start();
        new Thread(r, "Worker Thread").start();
    }
}
```
catch (InterruptedException ie)
{
}
System.out.println("Worker thread is dying");
};

Thread thd = new Thread(r);
thd.start();
System.out.println("Default main thread is doing work.");
try
{
    Thread.sleep(2000);
}
catch (InterruptedException ie)
{
}
System.out.println("Default main thread has finished its work.");
System.out.println("Default main thread is waiting for worker thread to die.");
try
{
    thd.join();
}
catch (InterruptedException ie)
{
}
System.out.println("Main thread is dying");
}
}

Listing 7-15 demonstrates the default main thread starting a worker thread, performing some work, and then waiting for the worker thread to die by calling \texttt{join()} via the worker thread’s \texttt{thd} object. When you run this application, you will discover output similar to the following (message order might differ somewhat):

Default main thread is doing work.
Worker thread is simulating work by sleeping for 5 seconds.
Default main thread has finished its work.
Default main thread is waiting for worker thread to die.
Worker thread is dying
Main thread is dying

Every Thread object belongs to some ThreadGroup object; Thread declares a ThreadGroup \texttt{getThreadGroup()} method that returns this object. You should ignore thread groups because they are not that useful. If you need to logically group Thread objects, you should use an array or collection instead.
Caution  Various ThreadGroup methods are flawed. For example, int enumerate(Thread[] threads) will not include all active threads in its enumeration when its threads array argument is too small to store their Thread objects. Although you might think that you could use the return value from the int activeCount() method to properly size this array, there is no guarantee that the array will be large enough because activeCount()'s return value fluctuates with the creation and death of threads.

However, you should still know about ThreadGroup because of its contribution in handling exceptions that are thrown while a thread is executing. Listing 7-16 sets the stage for learning about exception handling by presenting a run() method that attempts to divide an integer by 0, which results in a thrown ArithmeticException instance.

Listing 7-16.  Throwing an Exception from the run() Method

```
public class ExceptionThread
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                int x = 1 / 0; // Line 10
            }
        };
        Thread thd = new Thread(r);
        thd.start();
    }
}
```

Run this application and you will see an exception trace that identifies the thrown ArithmeticException.

```
Exception in thread "Thread-0" java.lang.ArithmeticException: / by zero
at ExceptionThread$1.run(ExceptionThread.java:10)
at java.lang.Thread.run(Unknown Source)
```

When an exception is thrown out of the run() method, the thread terminates and the following activities take place:

- The virtual machine looks for an instance of Thread.UncaughtExceptionHandler installed via Thread's void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) method. When this handler is found, it passes execution to the instance's void uncaughtException(Thread t, Throwable e) method, where t identifies the Thread object of the thread that threw the exception, and e identifies the thrown exception or error—perhaps a java.lang.OutOfMemoryError instance was thrown. If this method throws an exception/error, the exception/error is ignored by the virtual machine.
Assuming that setUncaughtExceptionHandler() was not called to install a handler, the virtual machine passes control to the associated ThreadGroup object's uncaughtException(Thread t, Throwable e) method. Assuming that ThreadGroup was not extended and that its uncaughtException() method was not overridden to handle the exception, uncaughtException() passes control to the parent ThreadGroup object's uncaughtException() method when a parent ThreadGroup is present. Otherwise, it checks to see if a default uncaught exception handler has been installed (via Thread's static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) method). If a default uncaught exception handler has been installed, its uncaughtException() method is called with the same two arguments. Otherwise, uncaughtException() checks its Throwable argument to determine if it is an instance of java.lang.ThreadDeath. If so, nothing special is done. Otherwise, as Listing 7-16's exception message shows, a message containing the thread's name, as returned from the thread's getName() method, and a stack backtrace, using the Throwable argument's printStackTrace() method, is printed to the standard error stream.

Listing 7-17 demonstrates Thread's setUncaughtExceptionHandler() and setDefaultUncaughtExceptionHandler() methods.

Listing 7-17. Demonstrating Uncaught Exception Handlers

```java
public class ExceptionThread
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                int x = 1 / 0;
            }
        }
        Thread thd = new Thread(r);
        Thread.UncaughtExceptionHandler uceh = new Thread.UncaughtExceptionHandler()
        {
            @Override
            public void uncaughtException(Thread t, Throwable e)
            {
                System.out.println("Caught throwable " + e + " for thread " + t);
            }
        };
        thd.setUncaughtExceptionHandler(uceh);
        uceh = new Thread.UncaughtExceptionHandler()
        {...
    }
```
{  
    @Override  
    public void uncaughtException(Thread t, Throwable e)  
    {  
        System.out.println("Default uncaught exception handler");  
        System.out.println("Caught throwable " + e + " for thread "+ t);  
    }  
}  

thd.setDefaultUncaughtExceptionHandler(uceh);  
thd.start();

When you run this application, you will observe the following output:

Caught throwable java.lang.ArithmeticException: / by zero for thread Thread[Thread-0,5,main]

You will not also see the default uncaught exception handler's output because the default handler is not called. To see that output, you must comment out thd.setUncaughtExceptionHandler(uceh);. If you also comment out thd.setDefaultUncaughtExceptionHandler(uceh);, you will see Listing 7-16's output.

Caution  Thread declares several deprecated methods, including stop() (stop an executing thread). These methods have been deprecated because they are unsafe. Do not use these deprecated methods. (I will show you how to safely stop a thread later in this chapter.) Also, you should avoid the static void yield() method, which is intended to switch execution from the current thread to another thread, because it can affect portability and hurt application performance. Although yield() might switch to another thread on some platforms (which can improve performance), yield() might only return to the current thread on other platforms (which hurts performance because the yield() call has only wasted time).

Synchronization

Throughout its execution, each thread is isolated from other threads because it has been given its own method-call stack. However, threads can still interfere with each other when they access and manipulate shared data. This interference can corrupt the shared data, and this corruption can cause an application to fail.

For example, consider a checking account in which a husband and wife have joint access. Suppose that the husband and wife decide to empty this account at the same time without knowing that the other is doing the same thing. Listing 7-18 demonstrates this scenario.
Listing 7-18. A Problematic Checking Account

```java
class CheckingAccount
{
    private int balance;

    public CheckingAccount(int initialBalance)
    {
        balance = initialBalance;
    }

    public boolean withdraw(int amount)
    {
        if (amount <= balance)
        {
            try
            {
                Thread.sleep((int) (Math.random() * 200));
            }
            catch (InterruptedException ie)
            {
            }
            balance -= amount;
            return true;
        }
        return false;
    }

    public static void main(String[] args)
    {
        final CheckingAccount ca = new CheckingAccount(100);
        Runnable r = new Runnable()
        {
            public void run()
            {
                String name = Thread.currentThread().getName();
                for (int i = 0; i < 10; i++)
                    System.out.println (name + " withdraws $10: " +
                                        ca.withdraw(10));
            }
        };
        Thread thdHusband = new Thread(r);
        thdHusband.setName("Husband");
        Thread thdWife = new Thread(r);
        thdWife.setName("Wife");
        thdHusband.start();
        thdWife.start();
    }
}
```
This application lets more money be withdrawn than is available in the account. For example, the following output reveals $110 being withdrawn when only $100 is available:

Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Wife withdraws $10: true
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false

The reason why more money is withdrawn than is available for withdrawal is that a race condition exists between the husband and wife threads.

Note A race condition is a scenario in which multiple threads are accessing shared data and the final result of these accesses is dependent on the timing of how the threads are scheduled. Race conditions can lead to bugs that are hard to find and results that are unpredictable.

In this example, there is a race condition between checking the amount for withdrawal to ensure that it is less than what appears in the balance and deducting the amount from the balance. The race condition exists because these actions are not atomic (indivisible) operations. (Although atoms are divisible, atomic is commonly used to refer to something being indivisible.)

Note The Thread.sleep() method call that sleeps for a variable amount of time (up to a maximum of 199 milliseconds) is present so that you can observe more money being withdrawn than is available for withdrawal. Without this method call, you might have to execute the application hundreds of times (or more) to witness this problem, because the scheduler might rarely pause a thread between the amount \( \leq \) balance expression and the balance \( -= \) amount; expression statement; the code executes rapidly.
Consider the following scenario:

- The Husband thread executes withdraw()’s amount <= balance expression, which returns true. The scheduler then suspends the Husband thread and executes the Wife thread.
- The Wife thread executes withdraw()’s amount <= balance expression, which returns true.
- The Wife thread performs the withdrawal. The scheduler suspends the Wife thread and resumes the Husband thread.
- The Husband thread performs the withdrawal.

This problem can be corrected by synchronizing access to withdraw() so that only one thread at a time can execute inside this method. You can synchronize access to this method by adding reserved word synchronized to the method header prior to the method’s return type, for example, synchronized boolean withdraw(int amount).

When you run the modified CheckingAccount application, you should observe output similar to the following:

```
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Wife withdraws $10: true
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
```

The need for synchronization is often subtle. For example, Listing 7-19’s ID utility class declares a getNextID() method that returns a unique long-based ID, perhaps to be used when generating unique filenames. Although you might not think so, this method can cause data corruption and return duplicate values on a 32-bit machine.
CHAPTER 7: Exploring the Basic APIs, Part 1

Listing 7-19. A Utility Class for Returning Unique IDs

class ID {
    private static long nextID = 0;
    static long getNextID() {
        return nextID++;
    }
}

Data corruption occurs because 32-bit virtual machine implementations require two steps to update a 64-bit long integer and adding 1 to nextID is not atomic: the scheduler could interrupt a thread that has only updated half of nextID, which corrupts the contents of this variable.

Note Variables of type long and double are subject to corruption when being written to in an unsynchronized context on 32-bit virtual machines. This problem doesn’t occur with variables of type boolean, byte, char, float, int, or short; each type occupies 32 bits or less.

Duplicate values are returned because postincrement (+++) reads and writes the nextID field in two steps, thread A reads nextID but does not increment its value before being interrupted by the scheduler, and thread B executes and reads the same value.

Both problems can be corrected by synchronizing access to nextID so that only one thread can execute this method’s code. All that is required is to add synchronized to the method header prior to the method’s return type; for example, static synchronized int getNextID().

As I will demonstrate later, you can also synchronize access to a block of statements by specifying the following syntax:

synchronized(object) {
    /* statements */
}

According to this syntax, object is an arbitrary object reference.

Mutual Exclusion, Monitors, and Locks

Whether you are synchronizing access to a method or a block of statements, no thread can enter the synchronized region until a thread that’s already executing inside that region leaves it. This property of synchronization is known as mutual exclusion.

Mutual exclusion is implemented in terms of monitors and locks. A monitor is a concurrency construct for controlling access to a critical section, a region of code that must execute atomically. It is identified at the source code level as a synchronized method or a synchronized block.
A lock is a token that a thread must acquire before a monitor allows that thread to execute inside a monitor’s critical section. The token is released automatically when the thread exits the monitor, to give another thread an opportunity to acquire the token and enter the monitor.

Note A thread that has acquired a lock doesn’t release this lock when it calls one of Thread’s sleep() methods.

A thread entering a synchronized instance method acquires the lock associated with the object on which the method is called. A thread entering a synchronized class method acquires the lock associated with the class’s java.lang.Class object. Finally, a thread entering a synchronized block acquires the lock associated with the block’s controlling object.

Tip Thread declares a static boolean holdsLock(Object o) method that returns true when the calling thread holds the monitor lock on object o. You will find this method handy in assertion statements, such as assert Thread.holdsLock(o);

Visibility

For performance reasons, each thread can have its own copy of a shared variable stored in a local cache (localized high-speed memory). Without synchronization, one thread’s write to its copy will not be visible to other thread’s copies. Ideally, when a thread updates a shared variable, this update should be made to the copy stored in main memory so that other threads can see these updates.

Synchronization also has the property of visibility in which shared variable values in main memory are copied to cache memory upon entry to a critical section and copied from cache memory to main memory upon exit from a critical section. Visibility addresses the vagaries of memory caching and compiler optimizations that might otherwise prevent one thread from observing another thread’s update of a shared variable.

Visibility makes it possible to communicate between threads. For example, you might design your own mechanism for stopping a thread (because you cannot use Thread’s unsafe stop() methods for this task). Listing 7-20 shows how you might accomplish this task.

Listing 7-20. Attempting to Stop a Thread

```java
public class ThreadStopping
{
    public static void main(String[] args)
    {
        class StoppableThread extends Thread
        {
            private boolean stopped = false;
        }
    }
```
@Override
public void run()
{
    while(!stopped)
        System.out.println("running");
}

void stopThread()
{
    stopped = true;
}
}  
StoppableThread thd = new StoppableThread();
thd.start();
try
{
    Thread.sleep(1000); // sleep for 1 second
}  
catch (InterruptedException ie)
{

}  
thd.stopThread();
}

Listing 7-20 introduces a main() method with a local class named StoppableThread that subclasses Thread. StoppableThread declares a stopped field initialized to false, a stopThread() method that sets this field to true, and a run() method whose infinite loop checks stopped on each loop iteration to see if its value has changed to true.

After instantiating StoppableThread, the default main thread starts the thread associated with this Thread object. It then sleeps for one second and calls StoppableThread’s stop() method before dying. When you run this application on a single-processor/single-core machine, you will probably observe the application stopping. You might not see this stoppage when the application runs on a multiprocessor machine or a uniprocessor machine with multiple cores where each processor or core probably has its own cache with its own copy of stopped. When one thread modifies its copy of this field, the other thread’s copy of stopped isn’t changed.

Listing 7-21 refactors Listing 7-20 to guarantee that the application will run correctly on all kinds of machines.

Listing 7-21. Guaranteed Stoppage on a Multiprocessor/Multicore Machine

public class ThreadStopping
{
    public static void main(String[] args)
    {
        class StoppableThread extends Thread
        {
            private boolean stopped = false;

            public void run()
            {
                while(!stopped)
                    System.out.println("running");
            }

            void stopThread()
            {
                stopped = true;
            }
        }
    
        StoppableThread thd = new StoppableThread();
        thd.start();
        try
        {
            Thread.sleep(1000); // sleep for 1 second
        }
        catch (InterruptedException ie)
        {
        }
        thd.stopThread();
    }
}
Listing 7-21’s stopThread() and isStopped() methods are synchronized to support visibility so that the default main thread that calls stopThread() and the started thread that executes inside run() can communicate. When a thread enters one of these methods, it’s guaranteed to access a single shared copy of the stopped field (not a cached copy).

Synchronization gives us mutual exclusion and visibility. Because you don’t need mutual exclusion in this example (there is no race condition), you can refactor Listing 7-21 to take advantage of Java’s volatile reserved word, which supports visibility only, and which Listing 7-22 demonstrates.

Listing 7-22. The volatile Alternative to Synchronization

```java
public class ThreadStopping{
    public static void main(String[] args){
        class StoppableThread extends Thread{
            private volatile boolean stopped = false;

            @Override
            public void run(){
                while(!isStopped())
                    System.out.println("running");
            }

            synchronized void stopThread(){
                stopped = true;
            }

            private synchronized boolean isStopped(){
                return stopped;
            }
        }StoppableThread thd = new StoppableThread();
thd.start();
try{
    Thread.sleep(1000); // sleep for 1 second
} catch (InterruptedException ie) {
    
}thd.stopThread();
}
}
```
{ 
    while(!stopped) 
    System.out.println("running"); 
} 

void stopThread() 
{ 
    stopped = true; 
} 

StoppableThread thd = new StoppableThread(); 
thd.start(); 
try 
{ 
    Thread.sleep(1000); // sleep for 1 second 
} 
catch (InterruptedException ie) 
{ 
} 

thd.stopThread(); 
}

Listing 7-22 declares stopped to be volatile. Threads that access this field will always access 
a single shared copy (not cached copies on multiprocessor/m multicore machines).

When a field is declared volatile, it cannot also be declared final. If you’re depending on the 
semantics (meaning) of volatility, you still get those from a final field. For more information, check 
out Brian Goetz’s “Java theory and practice: Fixing the Java Memory Model, Part 2” article 
(www.ibm.com/developerworks/library/j-jtp03304/).

**Caution**  Use volatile only in a thread communication context. Also, you can only use this reserved word 
in the context of field declarations. Although you can declare double and long fields volatile, you should 
avoid doing so on 32-bit virtual machines because it takes two operations to access a double or long 
variable’s value, and mutual exclusion (via synchronization) is required to access their values safely.

**Waiting and Notification**

The java.lang.Object class provides wait(), notify(), and notifyAll() methods to support a 
form of thread communication where a thread voluntarily waits for some condition (a prerequisite 
for continued execution) to arise, at which time another thread notifies the waiting thread that it 
can continue. wait() causes its calling thread to wait on an object’s monitor, and notify() and 
notifyAll() wake up one or all threads waiting on the monitor.
Caution Because the wait(), notify(), and notifyAll() methods depend on a lock, they cannot be called from outside of a synchronized method or synchronized block. If you fail to heed this warning, you will encounter a thrown instance of the java.lang.IllegalMonitorStateException class. Also, a thread that has acquired a lock releases this lock when it calls one of Object's wait() methods.

A classic example of thread communication involving conditions is the relationship between a producer thread and a consumer thread. The producer thread produces data items to be consumed by the consumer thread. Each produced data item is stored in a shared variable.

Imagine that the threads are running at different speeds. The producer might produce a new data item and record it in the shared variable before the consumer retrieves the previous data item for processing. Also, the consumer might retrieve the contents of the shared variable before a new data item is produced.

To overcome those problems, the producer thread must wait until it is notified that the previously produced data item has been consumed, and the consumer thread must wait until it is notified that a new data item has been produced. Listing 7-23 shows you how to accomplish this task via wait() and notify().

Listing 7-23. The Producer-Consumer Relationship, Version 1

```java
public class PC {
    public static void main(String[] args) {
        Shared s = new Shared();
        new Producer(s).start();
        new Consumer(s).start();
    }
}

class Shared {
    private char c = ' ';
    private boolean writeable = true;

    synchronized void setSharedChar(char c) {
        while (!writeable)
            try {
                wait();
            } catch (InterruptedException e) {}
        this.c = c;
    }
}
```

www.it-ebooks.info
writeable = false;
notify();
}

synchronized char getSharedChar()
{
    while (writeable)
        try
            {
            wait();
        }
        catch (InterruptedException e) {}
    writeable = true;
    notify();
    return c;
}

class Producer extends Thread
{
    private Shared s;

    Producer(Shared s)
    {
        this.s = s;
    }

    @Override
    public void run()
    {
        for (char ch = 'A'; ch <= 'Z'; ch++)
            {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
        }
    }
}

class Consumer extends Thread
{
    private Shared s;

    Consumer(Shared s)
    {
        this.s = s;
    }

    @Override
    public void run()
    {
        char ch;
        do
This application creates a Shared object and two threads that get a copy of the object's reference. The producer calls the object’s setSharedChar() method to save each of 26 uppercase letters; the consumer calls the object’s getSharedChar() method to acquire each letter.

The writeable instance field tracks two conditions: the producer waiting on the consumer to consume a data item and the consumer waiting on the producer to produce a new data item. It helps coordinate execution of the producer and consumer. The following scenario, where the consumer executes first, illustrates this coordination:

1. The consumer executes s.getSharedChar() to retrieve a letter.
2. Inside of that synchronized method, the consumer calls wait() because writeable contains true. The consumer now waits until it receives notification from the producer.
3. The producer eventually executes s.setSharedChar(ch);
4. When the producer enters that synchronized method (which is possible because the consumer released the lock inside of the wait() method prior to waiting), the producer discovers writeable’s value to be true and doesn’t call wait().
5. The producer saves the character, sets writeable to false (which will cause the producer to wait on the next setSharedChar() call when the consumer has not consumed the character by that time), and calls notify() to awaken the consumer (assuming the consumer is waiting).
6. The producer exits setSharedChar(char c).
7. The consumer wakes up (and reacquires the lock), sets writeable to true (which will cause the consumer to wait on the next getSharedChar() call when the producer has not produced a character by that time), notifies the producer to awaken that thread (assuming the producer is waiting), and returns the shared character.

Although the synchronization works correctly, you might observe output (on some platforms) that shows multiple producing messages before multiple consuming messages. For example, you might see A produced by producer., followed by B produced by producer., followed by A consumed by consumer., at the beginning of the application’s output.

This strange output order is caused by the call to setSharedChar() followed by its companion System.out.println() method call not being atomic, and by the call to getSharedChar() followed by its companion System.out.println() method call not being atomic. The output order can be
corrected by wrapping each of these method call pairs in a synchronized block that synchronizes on the Shared object referenced by s. Listing 7-24 presents this enhancement.

Listing 7-24. The Producer-Consumer Relationship, Version 2

```java
class Shared {
    private char c = '\u0000';
    private boolean writeable = true;

    synchronized void setSharedChar(char c) {
        while (!writeable)
            try {
                wait();
            } catch (InterruptedException e) {} 
        this.c = c;
        writeable = false;
        notify();
    }

    synchronized char getSharedChar() {
        while (writeable)
            try {
                wait();
            } catch (InterruptedException e) {} 
        writeable = true;
        notify();
        return c;
    }
}

class Producer extends Thread {
    private Shared s;
```
Producer(Shared s)
{
    this.s = s;
}

@Override
public void run()
{
    for (char ch = 'A'; ch <= 'Z'; ch++)
    {
        synchronized(s)
        {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
        }
    }
}

class Consumer extends Thread
{
    private Shared s;

    Consumer(Shared s)
    {
        this.s = s;
    }

    @Override
    public void run()
    {
        char ch;
        do
        {
            synchronized(s)
            {
                ch = s.getSharedChar();
                System.out.println(ch + " consumed by consumer.");
            }
        }
        while (ch != 'Z');
    }
}

Compile Listing 7-24 (javac PC.java) and run this application (java PC). Its output should always appear in the same alternating order as shown next (only the first few lines are shown for brevity):

A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.

**Caution**  Never call `wait()` outside of a loop. The loop tests the condition (!writeable or writeable in the previous example) before and after the `wait()` call. Testing the condition before calling `wait()` ensures *liveness*. If this test was not present, and if the condition held and `notify()` had been called prior to `wait()` being called, it is unlikely that the waiting thread would ever wake up. Retesting the condition after calling `wait()` ensures *safety*. If retesting didn’t occur, and if the condition didn’t hold after the thread had awakened from the `wait()` call (perhaps another thread called `notify()` accidentally when the condition didn’t hold), the thread would proceed to destroy the lock’s protected invariants.

**Deadlock**

Too much synchronization can be problematic. If you are not careful, you might encounter a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section. Listing 7-25’s atypical example demonstrates this scenario, which is known as *deadlock*.

**Listing 7-25. A Pathological Case of Deadlock**

```java
public class DeadlockDemo
{
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void instanceMethod1()
    {
        synchronized(lock1)
        {
            synchronized(lock2)
            {
                System.out.println("first thread in instanceMethod1");
                // critical section guarded first by
                // lock1 and then by lock2
            }
        }
    }

    public void instanceMethod2()
    {
        synchronized(lock2)
        {
            synchronized(lock1)
            {
                System.out.println("second thread in instanceMethod2");
            }
        }
    }
}
```
public static void main(String[] args) {
    final DeadlockDemo dld = new DeadlockDemo();
    Runnable r1 = new Runnable()
    {
        @Override
        public void run()
        {
            while(true)
            {
                dld.instanceMethod1();
                try
                {
                    Thread.sleep(50);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    };
    Thread thdA = new Thread(r1);
    Runnable r2 = new Runnable()
    {
        @Override
        public void run()
        {
            while(true)
            {
                dld.instanceMethod2();
                try
                {
                    Thread.sleep(50);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    };
    Thread thdB = new Thread(r2);
    thdA.start();
    thdB.start();
}
Listing 7-25's thread A and thread B call `instanceMethod1()` and `instanceMethod2()`, respectively, at different times. Consider the following execution sequence:

1. Thread A calls `instanceMethod1()`, obtains the lock assigned to the `lock1`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock2`-referenced object).

2. Thread B calls `instanceMethod2()`, obtains the lock assigned to the `lock2`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock1`-referenced object).

3. Thread A attempts to acquire the lock associated with `lock2`. The virtual machine forces the thread to wait outside of the inner critical section because thread B holds that lock.

4. Thread B attempts to acquire the lock associated with `lock1`. The virtual machine forces the thread to wait outside of the inner critical section because thread A holds that lock.

5. Neither thread can proceed because the other thread holds the needed lock. You have a deadlock situation and the program (at least in the context of the two threads) freezes up.

Although the previous example clearly identifies a deadlock state, it’s often not that easy to detect deadlock. For example, your code might contain the following circular relationship among various classes (in several source files):

- Class A's synchronized method calls class B's synchronized method.
- Class B’s synchronized method calls class C’s synchronized method.
- Class C’s synchronized method calls class A's synchronized method.

If thread A calls class A’s synchronized method and thread B calls class C’s synchronized method, thread B will block when it attempts to call class A’s synchronized method and thread A is still inside of that method. Thread A will continue to execute until it calls class C’s synchronized method, and then block. Deadlock is the result.

Note Neither the Java language nor the virtual machine provides a way to prevent deadlock, and so the burden falls on you. The simplest way to prevent deadlock from happening is to avoid having either a synchronized method or a synchronized block call another synchronized method/block. Although this advice prevents deadlock from happening, it is impractical because one of your synchronized methods/blocks might need to call a synchronized method in a Java API, and the advice is overkill because the synchronized method/block being called might not call any other synchronized method/block, so deadlock would not occur.
Thread-Local Variables

You will sometimes want to associate per-thread data (such as a user ID) with a thread. Although you can accomplish this task with a local variable, you can only do so while the local variable exists. You could use an instance field to keep this data around longer, but then you would have to deal with synchronization. Thankfully, Java supplies ThreadLocal as a simple (and very handy) alternative.

Each instance of the ThreadLocal class describes a **thread-local variable**, which is a variable that provides a separate storage slot to each thread that accesses the variable. You can think of a thread-local variable as a multislot variable in which each thread can store a different value in the same variable. Each thread sees only its value and is unaware of other threads having their own values in this variable.

ThreadLocal is generically declared as ThreadLocal<T>, where T identifies the type of value that is stored in the variable. This class declares the following constructor and methods:

- ThreadLocal() creates a new thread-local variable.
- T get() returns the value in the calling thread's storage slot. If an entry doesn't exist when the thread calls this method, get() calls initialValue().
- T initialValue() creates the calling thread's storage slot and stores an initial (default) value in this slot. The initial value defaults to null. You must subclass ThreadLocal and override this protected method to provide a more suitable initial value.
- void remove() removes the calling thread's storage slot. If this method is followed by get() with no intervening set(), get() calls initialValue().
- void set(T value) sets the value of the calling thread's storage slot to value.

Listing 7-26 shows how to use ThreadLocal to associate different user IDs with two threads.

**Listing 7-26. Different User IDs for Different Threads**

```java
public class ThreadLocalDemo {
    private static volatile ThreadLocal<String> userID =
            new ThreadLocal<String>();

    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                if (name.equals("A"))
                    userID.set("foxtrot");
                else
                    userID.set("charlie");
                System.out.println(name + " " + userID.get());
            }
        };
```
Thread thdA = new Thread(r);
thdA.setName("A");
Thread thdB = new Thread(r);
thdB.setName("B");
 thdA.start();
 thdB.start();
}
}

After instantiating `ThreadLocal` and assigning the reference to a volatile class field named `userID` (the field is volatile because it is accessed by different threads, which might execute on a multiprocessor/multicore machine), the default main thread creates two more threads that store different `String` objects in `userID` and output their objects.

When you run this application, you will observe the following output (possibly not in this order):

A foxtrot
B charlie

Values stored in thread-local variables are not related. When a new thread is created, it gets a new storage slot containing `initialValue()`’s value. Perhaps you would prefer to pass a value from a `parent thread`, a thread that creates another thread, to a `child thread`, the created thread. You accomplish this task with `InheritableThreadLocal`.

`InheritableThreadLocal` is a subclass of `ThreadLocal`. As well as declaring an `InheritableThreadLocal()` constructor, this class declares the following protected method:

- `T childValue(T parentValue)` calculates the child’s initial value as a function of the parent’s value at the time the child thread is created. This method is called from the parent thread before the child thread is started. The method returns the argument passed to `parentValue` and should be overridden when another value is desired.

Listing 7-27 shows how to use `InheritableThreadLocal` to pass a parent thread’s `Integer` object to a child thread.

**Listing 7-27. Passing an Object from Parent Thread to Child Thread**

```java
public class InheritableThreadLocalDemo
{
    private static volatile InheritableThreadLocal<Integer> intVal =
    new InheritableThreadLocal<Integer>();

    public static void main(String[] args)
    {
        Runnable rP = new Runnable()
        {
            @Override
            public void run()
            {
                intVal.set(new Integer(10));
                Runnable rC = new Runnable()
```
```java
{  
    @Override
    public void run()
    {
        Thread thd;
        thd = Thread.currentThread();
        String name = thd.getName();
        System.out.println(name + " " + intVal.get());
    }
}
Thread thdChild = new Thread(rC);
thdChild.setName("Child");
thdChild.start();
}
new Thread(rP).start();
}
```

After instantiating `InheritableThreadLocal` and assigning it to a volatile class field named `intVal`, the default main thread creates a parent thread, which stores an `Integer` object containing 10 in `intVal`. The parent thread creates a child thread, which accesses `intVal` and retrieves its parent thread’s `Integer` object.

When you run this application, you will observe the following output:

```
Child 10
```

**Note** For more insight into `ThreadLocal` and how it is implemented, check out Patson Luk’s “A Painless Introduction to Java's ThreadLocal Storage” blog post [http://java.dzone.com/articles/painless-introduction-javas-threadlocal-storage](http://java.dzone.com/articles/painless-introduction-javas-threadlocal-storage).

## EXERCISES

The following exercises are designed to test your understanding of Chapter 7’s content.

1. What constants does `Math` declare?
2. Why is `Math.abs(Integer.MIN_VALUE)` equal to `Integer.MIN_VALUE`?
4. Identify the five special values that can arise during floating-point calculations.
5. How do `Math` and `StrictMath` differ?
6. What is the purpose of `strictfp`?
7. What is BigDecimal and why might you use this class?
8. Which RoundingMode constant describes the form of rounding commonly taught at school?
9. What is BigInteger?
10. What is a primitive type wrapper class?
11. Identify Java's primitive type wrapper classes.
12. Why does Java provide primitive type wrapper classes?
13. True or false: Byte is the smallest of the primitive type wrapper classes.
14. Why should you use Character class methods instead of expressions such as ch >= '0' && ch <= '9' to determine whether or not a character is a digit, a letter, and so on?
15. How do you determine whether or not double variable d contains +infinity or -infinity?
16. Identify the class that is the superclass of Byte, Character, and the other primitive type wrapper classes.
17. True or false: A string literal is a String object.
18. What is the purpose of String's intern() method?
19. How do String and StringBuffer differ?
20. How do StringBuffer and StringBuilder differ?
21. What System method do you invoke to copy an array to another array?
22. What System method do you invoke to obtain the current time in milliseconds?
23. Define thread.
24. What is the purpose of the Runnable interface?
25. What is the purpose of the Thread class?
26. True or false: A Thread object associates with multiple threads.
27. Define race condition.
28. What is synchronization?
29. How is synchronization implemented?
30. How does synchronization work?
31. True or false: Variables of type long or double are not atomic on 32-bit virtual machines.
32. What is the purpose of reserved word volatile?
33. True or false: Object's wait() methods can be called from outside of a synchronized method or block.
34. Define deadlock.
35. What is the purpose of the ThreadLocal class?
36. How does InheritableThreadLocal differ from ThreadLocal?
37. A prime number is a positive integer greater than 1 that is evenly divisible only by 1 and itself. Create a PrimeNumberTest application that determines if its solitary integer argument is prime or not prime, and outputs a suitable message. For example, java PrimeNumberTest 289 should output the message 289 is not prime. A simple way to check for primality is to loop from 2 through the square root of the integer argument, and use the remainder operator in the loop to determine if the argument is divided evenly by the loop index. For example, because 6 % 2 yields a remainder of 0 (2 divides evenly into 6), integer 6 is not a prime number.

38. Create a MultiPrint application that takes two arguments: text and an integer value that represents a count. This application should print count copies of the text, one copy per line.

39. Rewrite the following inefficient loop to use StringBuffer. The resulting loop should minimize object creation:

```java
String[] imageNames = new String[NUM_IMAGES];
for (int i = 0; i < imageNames.length; i++)
    imageNames[i] = new String("image" + i + ".png");
```

40. Create a DigitsToWords application that accepts a single integer-based command-line argument. This application converts this argument to an int value (via Integer.parseInt(args[0])) and then passes the result to a String convertDigitsToWords(int integer) class method that returns a string containing a textual representation of that number. For example, 1 converts to one, 16 converts to sixteen, 69 converts to sixty-nine, 123 converts to one hundred and twenty-three, and 2938 converts to two thousand nine hundred and thirty-eight. Throw java.lang.IllegalArgumentException when the value passed to integer is less than 0 or greater than 9999. Use the StringBuffer class to serve as a repository for the generated text. Usage example: java DigitsToWords 2938.

41. Create an EVDump application that dumps all environment variables (not system properties) to the standard output.

42. Modify Listing 7-13's CountingThreads application by marking the two started threads as daemon threads. What happens when you run the resulting application?

43. Modify Listing 7-13's CountingThreads application by adding logic to stop both counting threads when the user presses the Return/Enter key. The default main thread of the new StopCountingThreads application should call System.in.read() before terminating, and assign true to a variable named stopped after this method call returns. At each loop iteration start, each counting thread should test this variable to see if it contains true, and only continue the loop when the variable contains false.
Summary

The standard class library offers many basic APIs via its java.lang and other packages. For example, the Math class supplements the basic math operations (+, -, *, /, and %) with advanced operations (such as trigonometry). The companion StrictMath class ensures that all of these operations yield the same values on all platforms.

The abstract java.lang.Number class is the superclass of those classes representing numeric values that are convertible to the byte integer, double precision floating-point, floating-point, integer, long integer, and short integer primitive types.

Money must never be represented by floating-point and double precision floating-point variables because not all monetary values can be represented exactly. In contrast, Number's BigDecimal subclass lets you accurately represent and manipulate these values.

BigDecimal relies on Number's BigInteger subclass for representing its unscaled value. A BigInteger instance describes an integer value that can be of arbitrary length (subject to the limits of the virtual machine's memory).

Number's Boolean, Byte, Character, Double, Float, Integer, Long, and Short subclasses are known as primitive type wrapper classes or value classes because their instances wrap themselves around values of primitive types.

Java provides these eight primitive type wrapper classes so that primitive type values can be stored in collections, such as lists, sets, and maps. Furthermore, these classes provide a good place to associate useful constants and class methods with the primitive types.

String represents a string as a sequence of characters. Because String instances are immutable, Java provides StringBuffer and StringBuilder for building strings more efficiently. The former class is used in multithreaded contexts; the latter (and more performant) class is for single-threaded use.

Applications execute via threads that serve as independent paths of execution through an application's code. The virtual machine gives each thread its own method-call stack to prevent threads from interfering with each other.

Java supports threads via its Threads API. This API largely consists of one interface (Runnable) and four classes (Thread, ThreadGroup, ThreadLocal, and InheritableThreadLocal) in the java.lang package. ThreadGroup is not as useful as these other types.

Throughout its execution, each thread is isolated from other threads because it has been given its own method-call stack. However, threads can still interfere with each other when they access and manipulate shared data. This interference can corrupt the shared data, causing an application to fail.

Corruption can be avoided by using synchronization so that only one thread at a time can execute inside a critical section, a region of code that must execute atomically. It is identified at the source code level as a synchronized method or a synchronized block.

You synchronize access at the method level by adding reserved word synchronized to the method header prior to the method's return type. You can also synchronize access to a block of statements by specifying a synchronized block via the following syntax: synchronized(object) {
/* statements */
}. 
Synchronization supports mutual exclusion and is implemented in terms of monitors and locks. A monitor controls access to a critical section and a lock must be acquired by a thread before the monitor will allow the thread to execute inside the monitor’s critical section.

Synchronization also supports visibility in which a thread always sees the main memory value and not a cached value of a shared variable. There exists an alternative to synchronization when only visibility is required. This alternative is reserved word `volatile`.

Object’s `wait()`, `notify()`, and `notifyAll()` methods support a form of thread communication where a thread voluntarily waits for some condition to arise, at which time another thread notifies the waiting thread that it can continue. `wait()` causes its calling thread to wait on an object’s monitor, and `notify()` and `notifyAll()` wake up one or all threads waiting on the monitor.

Too much synchronization can be problematic. If you are not careful, you might encounter a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section. This scenario is known as deadlock.

You will sometimes want to associate per-thread data with a thread. Although you can accomplish this task with a local variable, you can only do so while the local variable exists. You could use an instance field to keep this data around longer, but then you would have to deal with synchronization. Java supplies the `ThreadLocal` class as a simple (and very handy) alternative.

Each `ThreadLocal` instance describes a thread-local variable that provides a separate storage slot to each thread that accesses the variable. Think of a thread-local variable as a multislot variable in which each thread can store a different value in the same variable. Each thread sees only its value and is unaware of other threads having their own values in this variable.

Values stored in thread-local variables are not related. When a new thread is created, it gets a new storage slot containing `initialValue()`’s value. However, you can pass a value from a parent thread to a child thread by working with the `InheritableThreadLocal` class.

Chapter 8 continues to explore the basic APIs by focusing on Random, References, Reflection, StringTokenizer, and Timer and TimerTask.
There are more basic APIs in the `java.lang` package and also in `java.lang.ref`, `java.lang.reflect`, and `java.util` to consider for your Android apps. For example, you can add timers to your games.

**Exploring Random**

In Chapter 7, I formally introduced you to the `java.lang.Math` class’s `random()` method. If you were to investigate this method’s source code from the perspective of Java 7, you would encounter the following implementation:

```java
private static Random randomNumberGenerator;

private static synchronized Random initRNG() {
    Random rnd = randomNumberGenerator;
    return (rnd == null) ? (randomNumberGenerator = new Random()) : rnd;
}

public static double random() {
    Random rnd = randomNumberGenerator;
    if (rnd == null) rnd = initRNG();
    return rnd.nextDouble();
}
```

This code excerpt shows you that Math’s `random()` method is implemented in terms of a class named `Random`, which is located in the `java.util` package. `Random` instances generate sequences of random numbers and are known as *random number generators*.
Note These numbers are not truly random because they are generated from a mathematical algorithm. As a result, they are often referred to as pseudorandom numbers. However, it is often convenient to drop the “pseudo” prefix and refer to them as random numbers. Also, delaying object creation (new Random(), for example) until the first time the object is needed is known as lazy initialization.

Random generates its sequence of random numbers by starting with a special 48-bit value that is known as a seed. This value is subsequently modified by a mathematical algorithm, which is known as a linear congruential generator.

Note Check out Wikipedia’s “Linear congruential generator” entry (http://en.wikipedia.org/wiki/Linear_congruential_generator) to learn about this algorithm for generating random numbers.

Random declares a pair of constructors:

- Random() creates a new random number generator. This constructor sets the seed of the random number generator to a value that is very likely to be distinct from any other call to this constructor.
- Random(long seed) creates a new random number generator using its seed argument. This argument is the initial value of the internal state of the random number generator, which is maintained by the protected int next(int bits) method.

Because Random() doesn’t take a seed argument, the resulting random number generator always generates a different sequence of random numbers. This explains why Math.random() generates a different sequence each time an application starts running.

Tip Random(long seed) gives you the opportunity to reuse the same seed value, allowing the same sequence of random numbers to be generated. You will find this capability useful when debugging a faulty application that involves random numbers.

Random(long seed) calls the void setSeed(long seed) method to set the seed to the specified value. If you call setSeed() after instantiating Random, the random number generator is reset to the state that it was in immediately after calling Random(long seed).

The previous code excerpt demonstrates Random’s double nextDouble() method, which returns the next pseudorandom, uniformly distributed double precision floating-point value between 0.0 and 1.0 in this random number generator’s sequence.

Random also declares the following methods for returning other kinds of values:

- boolean nextBoolean() returns the next pseudorandom, uniformly distributed Boolean value in this random number generator’s sequence. Values true and false are generated with (approximately) equal probability.
- `void nextBytes(byte[] bytes)` generates pseudorandom byte integer values and stores them in the `bytes` array. The number of generated bytes is equal to the length of the `bytes` array.

- `float nextFloat()` returns the next pseudorandom, uniformly distributed floating-point value between 0.0 and 1.0 in this random number generator's sequence.

- `double nextGaussian()` returns the next pseudorandom, Gaussian ("normally") distributed double precision floating-point value with mean 0.0 and standard deviation 1.0 in this random number generator's sequence.

- `int nextInt()` returns the next pseudorandom, uniformly distributed integer value in this random number generator's sequence. All 4,294,967,296 possible integer values are generated with (approximately) equal probability.

- `int nextInt(int n)` returns a pseudorandom, uniformly distributed integer value between 0 (inclusive) and the specified value (exclusive) drawn from this random number generator's sequence. All `n` possible integer values are generated with (approximately) equal probability.

- `long nextLong()` returns the next pseudorandom, uniformly distributed long integer value in this random number generator's sequence. Because `Random` uses a seed with only 48 bits, this method will not return all possible 64-bit long integer values.

The `java.util.Collections` class declares a pair of `shuffle()` methods for shuffling the contents of a list. In contrast, the `java.util.Arrays` class doesn't declare a `shuffle()` method for shuffling the contents of an array. Listing 8-1 addresses this omission.

**Listing 8-1. Shuffling an Array of Integers**

```java
import java.util.Random;

public class Shuffler {
    public static void main(String[] args) {
        Random r = new Random();
        int[] array = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        for (int i = 0; i < array.length; i++) {
            int n = r.nextInt(array.length);
            // swap array[i] with array[n]
            int temp = array[i];
            array[i] = array[n];
            array[n] = temp;
        }
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");
        System.out.println();
    }
}
```

www.it-ebooks.info
Listing 8-1 presents a simple recipe for shuffling an array of integers—this recipe could be generalized. For each array entry from the start of the array to the end of the array, this entry is swapped with another entry whose index is chosen by `nextInt(int n)`.

When you run this application, you will observe a shuffled sequence of integers that is similar to the following sequence that I observed:

```
9 0 5 6 2 3 8 4 1 7
```

**Exploring References**

Chapter 3 introduced you to garbage collection in which you learned that the garbage collector removes an object from the heap when there are no more references to the object. This statement isn't completely true, as you will shortly discover.

Chapter 4 introduced you to `Object`'s `finalize()` method in which you learned that the garbage collector calls this method before removing an object from the heap. The `finalize()` method gives the object an opportunity to perform cleanup.

This section continues from where Chapters 3 and 4 left off by introducing you to Java's References API. After acquainting you with some basic terminology, it introduces you to the API's `Reference` and `ReferenceQueue` classes, followed by the API's `SoftReference`, `WeakReference`, and `PhantomReference` classes. These classes let applications interact with the garbage collector in limited ways.


**Basic Terminology**

When an application runs, its execution reveals a *root set of references*, which is a collection of local variables, parameters, class fields, and instance fields that currently exist and that contain (possibly null) references to objects. This root set changes over time as the application runs. For example, parameters disappear after a method returns.

Many garbage collectors identify this root set when they run. They use the root set to determine if an object is *reachable* (referenced, also known as *live*) or *unreachable* (not referenced). The garbage collector cannot collect reachable objects. Instead, it can only collect objects that, starting from the root set of references, cannot be reached.

**Note**  Reachable objects include objects that are indirectly reachable from root-set variables, which means objects that are reachable through live objects that are directly reachable from those variables. An object that is unreachable by any path from any root-set variable is eligible for garbage collection.
CHAPTER 8: Exploring the Basic APIs, Part 2

Beginning with Java 1.2, reachable objects are classified as strongly reachable, softly reachable, weakly reachable, and phantom reachable. Unlike strongly reachable objects, softly, weakly, and phantom reachable objects can be garbage collected.

Going from strongest to weakest, the different levels of reachability reflect the life cycle of an object. They are defined as follows:

- An object is **strongly reachable** when it can be reached from some thread without traversing any Reference objects. A newly created object (such as the object referenced by `d` in `Double d = new Double(1.0);`) is strongly reachable by the thread that created it.

- An object is **softly reachable** when it's not strongly reachable but can be reached by traversing a *soft reference* (a reference to the object where the reference is stored in a SoftReference object). The strongest reference to this object is a soft reference. When the soft references to a softly reachable object are cleared, the object becomes eligible for finalization (discussed in Chapter 4).

- An object is **weakly reachable** when it's neither strongly reachable nor softly reachable but can be reached by traversing a *weak reference* (a reference to the object where the reference is stored in a WeakReference object). The strongest reference to this object is a weak reference. When the weak references to a weakly reachable object are cleared, the object becomes eligible for finalization. (Apart from the garbage collector being more eager to clean up the weakly reachable object, a weak reference is exactly like a soft reference.)

- An object is **phantom reachable** when it's neither strongly, softly, nor weakly reachable, it has been finalized, and it's referred to by some *phantom reference* (a reference to the object where the reference is stored in a PhantomReference object). The strongest reference to this object is a phantom reference.

- Finally, an object is unreachable and therefore eligible for removal from memory during the next garbage collection cycle when it's not reachable in any of the above ways.

The object whose reference is stored in a SoftReference, WeakReference, or PhantomReference object is known as a **referent**.

**Reference and ReferenceQueue**

The References API consists of five classes located in the java.lang.ref package. Central to this package are Reference and ReferenceQueue

- Reference is the abstract superclass of this package’s concrete SoftReference, WeakReference, and PhantomReference subclasses.

- ReferenceQueue is a concrete class whose instances describe queue data structures. When you associate a ReferenceQueue instance with a Reference subclass object (Reference object, for short), the Reference object is added to the queue when the referent to which its encapsulated reference refers becomes garbage.
Note  You associate a ReferenceQueue object with a Reference object by passing the ReferenceQueue object to an appropriate Reference subclass constructor.

Reference is declared as generic type Reference<T> where T identifies the referent’s type. This class provides the following methods:

- **void clear()** assigns null to the stored reference; the Reference object on which this method is called isn’t enqueued (inserted) into its associated reference queue (when there is an associated reference queue). (The garbage collector clears references directly; it doesn’t call clear(). Instead, this method is called by applications.)

- **boolean enqueue()** adds the Reference object on which this method is called to the associated reference queue. This method returns true when this Reference object has become enqueued; otherwise, this method returns false—this Reference object was already enqueued or was not associated with a queue when created. (The garbage collector enqueues Reference objects directly; it doesn’t call enqueue(). Instead, this method is called by applications.)

- **T get()** returns this Reference object’s stored reference. The return value is null when the stored reference has been cleared, either by the application or by the garbage collector.

- **boolean isEnqueued()** returns true when this Reference object has been enqueued, either by the application or by the garbage collector. Otherwise, this method returns false—this Reference object was not associated with a queue when created.

Note  Reference also declares constructors. Because these constructors are package-private, only classes in the java.lang.ref package can subclass Reference. This restriction is necessary because instances of Reference’s subclasses must work closely with the garbage collector.

ReferenceQueue is declared as generic type ReferenceQueue<T> where T identifies the referent’s type. This class declares the following constructor and methods:

- **ReferenceQueue()** initializes a new ReferenceQueue instance.

- **Reference<? extends T> poll()** polls this queue to check for an available Reference object. When one is available, the object is removed from the queue and returned. Otherwise, this method returns immediately with a null value.

- **Reference<? extends T> remove()** removes the next Reference object from the queue and returns this object. This method waits indefinitely for a Reference object to become available and throws java.lang.InterruptedException when this wait is interrupted.
Reference<? extends T> remove(long timeout) removes the next Reference object from the queue and returns this object. This method waits until a Reference object becomes available or until timeout milliseconds have elapsed—passing 0 to timeout causes the method to wait indefinitely. If timeout’s value expires, the method returns null. This method throws \texttt{java.lang.IllegalArgumentException} when timeout’s value is negative or \texttt{InterruptedException} when this wait is interrupted.

### SoftReference

The \texttt{SoftReference} class describes a Reference object whose referent is softly reachable. As well as inheriting Reference’s methods and overriding \texttt{get()}, this generic class provides the following constructors for initializing a \texttt{SoftReference} object:

- \texttt{SoftReference(T r)} encapsulates \texttt{r}’s reference. The \texttt{SoftReference} object behaves as a soft reference to \texttt{r}. No \texttt{ReferenceQueue} object is associated with this \texttt{SoftReference} object.
- \texttt{SoftReference(T r, ReferenceQueue<? super T> rq)} encapsulates \texttt{r}’s reference. The \texttt{SoftReference} object behaves as a soft reference to \texttt{r}. The \texttt{ReferenceQueue} object identified by \texttt{rq} is associated with this \texttt{SoftReference} object. Passing \texttt{null} to \texttt{rq} indicates a soft reference without a queue.

\texttt{SoftReference} is useful for implementing caches of objects that are expensive timewise to create (such as a database connection) and/or occupy significant amounts of heap space, such as large images. I’ll present an example when I discuss the Collection Framework’s \texttt{java.util.HashMap} class (in Chapter 9).

### WeakReference

The \texttt{WeakReference} class describes a Reference object whose referent is weakly reachable. As well as inheriting Reference’s methods, this generic class provides the following constructors for initializing a \texttt{WeakReference} object:

- \texttt{WeakReference(T r)} encapsulates \texttt{r}’s reference. The \texttt{WeakReference} object behaves as a weak reference to \texttt{r}. No \texttt{ReferenceQueue} object is associated with this \texttt{WeakReference} object.
- \texttt{WeakReference(T r, ReferenceQueue<? super T> rq)} encapsulates \texttt{r}’s reference. The \texttt{WeakReference} object behaves as a weak reference to \texttt{r}. The \texttt{ReferenceQueue} object identified by \texttt{rq} is associated with this \texttt{WeakReference} object. Passing \texttt{null} to \texttt{rq} indicates a weak reference without a queue.

\texttt{WeakReference} is useful for preventing memory leaks related to hashmaps and is used to implement the Collection Framework’s \texttt{java.util.WeakHashMap} class. I’ll have more to say about this topic when I discuss \texttt{WeakHashMap} (in Chapter 9).
PhantomReference

The PhantomReference class describes a Reference object whose referent is phantom reachable. As well as inheriting Reference’s methods and overriding get(), this generic class provides a single constructor for initializing a PhantomReference object:

```
PhantomReference(T r, ReferenceQueue<? super T> rq) encapsulates r’s reference. The PhantomReference object behaves as a phantom reference to r. The ReferenceQueue object identified by rq is associated with this PhantomReference object. Passing null to rq makes no sense because get() is overridden to return null and the PhantomReference object will never be enqueued.
```

Although you cannot access a PhantomReference object’s referent (its get() method returns null), this class is useful because enqueuing the PhantomReference object signals that the referent has been finalized and its memory space has been reclaimed. For example, you can learn that a large object has been removed from memory and then load another large object into memory without having to worry about an out-of-memory error.

Another use for PhantomReference is to avoid the following problem: an object’s finalize() method can resurrect the object by creating a new strong reference to the object. Because of this resurrection potential, the garbage collector requires at least two garbage collection cycles to determine if the object can be garbage collected. When the first cycle detects that the object is eligible for garbage collection, it calls finalize(). Because this method might perform resurrection (see Chapter 4), which makes the unreachable object reachable, a second garbage collection cycle is needed to determine if resurrection has happened. This extra cycle slows down garbage collection and has the potential to result in an out-of-memory error. If finalize() isn’t overridden, the garbage collector doesn’t need to call that method and considers the object to be finalized. Hence, the garbage collector requires only one cycle.

You can avoid the need for finalize() and its performance implications/out-of-memory error potential by using PhantomReference. Resurrection cannot happen because the reference queue’s get() method returns null. Listing 8-2 shows how you might use PhantomReference to detect the finalization of a large object.

Listing 8-2. Detecting a Large Object’s Finalization

```java
import java.lang.ref.PhantomReference;
import java.lang.ref.ReferenceQueue;

class LargeObject
{
    private byte[] memory = new byte[1024 * 1024 * 50]; // 50 megabytes
}

class PhantomReferenceDemo
{
    public static void main(String[] args)
    {
        ReferenceQueue<LargeObject> rq = new ReferenceQueue<LargeObject>();
        PhantomReference<LargeObject> pr;
        pr = new PhantomReference<LargeObject>(new LargeObject(), rq);
```
while (rq.poll() == null) {
    System.out.println("waiting for first large object to be finalized");
    new LargeObject(); // Create another (unreferenced) LargeObject.
}
System.out.println("first large object finalized");
System.out.println("pr.get() returns " + pr.get());
}

Listing 8-2 declares a LargeObject class whose private memory array occupies 50MB. If your virtual machine throws java.lang.OutOfMemoryError when you run this application, you might need to reduce the array’s size.

The main() method first creates a ReferenceQueue object describing a queue onto which a PhantomReference object that initially contains a LargeObject reference will be enqueued. It then creates the PhantomReference object, passing a reference to a newly created LargeObject instance and a reference to the previously created ReferenceQueue object to the constructor.

main() now enters a polling loop, which first calls poll() to detect the finalization of the LargeObject instance. As long as this method returns null, meaning that the LargeObject instance is still unfinalized, the loop outputs a message and creates a new unreferenced LargeObject instance.

At some point, the garbage collector will run. It will clear the PhantomReference object’s LargeObject reference and finalize the LargeObject object. The PhantomReference object is then enqueued onto the rq-referenced ReferenceQueue; poll() returns the PhantomReference object.

main() now exits the loop, outputs a message confirming the large object’s finalization, and outputs pr.get()’s return value, which is null—proving that you cannot access a PhantomReference object’s referent.

Compile Listing 8-2 (javac PhantomReferenceDemo.java) and run the application (java PhantomReferenceDemo). You should discover output that’s similar to the following output:
Exploring Reflection

Chapter 4 presented two forms of runtime type identification (RTTI). Java’s Reflection API offers a third RTTI form in which applications can dynamically load and learn about loaded classes and other reference types. The API also lets applications instantiate classes, call methods, access fields, and perform other tasks reflectively. This form of RTTI is known as reflection or introspection.

Caution Reflection should not be used indiscriminately. Application performance suffers because it takes longer to perform operations with reflection than without reflection. Also, reflection-oriented code can be harder to read, and the absence of compile-time type checking can result in runtime failures.

Chapter 6 presented a StubFinder application that used part of the Reflection API to load a class and identify all of the loaded class’s public methods that are annotated with @Stub annotations. This tool is one example where using reflection is beneficial. Another example is the class browser, a tool that enumerates the members of a class.

Note You’ll occasionally use the Reflection API in your Android app projects. For example, reflection simplifies the task of coding an app to use newer APIs when available. For more information, check out “Backward compatibility for Android applications” (http://android-developers.blogspot.ca/2009/04/backward-compatibility-for-android.html).

The Class Entry Point

The java.lang package’s Class class is the entry point into the Reflection API, whose types are stored mainly in the java.lang.reflect package. Class is generically declared as Class<T>, where T identifies the class, interface, enum, or annotation type that’s being modeled by the Class object. T can be replaced by ? (as in Class<?>>) when the type being modeled is unknown.
Table 8-1 describes some of Class’s methods.

**Table 8-1. Class Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static Class&lt;?&gt; forName(String typename)</code></td>
<td>Returns the Class object associated with typename, which must include the type’s qualified package name when the type is part of a package (java.lang.String, for example). If the class or interface type has not been loaded into memory, this method takes care of loading (reading the classfile’s contents into memory), linking (taking these contents and combining them into the runtime state of the virtual machine so that they can be executed), and initializing (setting class fields to default values, running class initializers, and performing other class initialization) before returning the Class object. This method throws java.lang.ClassNotFoundException when the type cannot be found, java.lang.LinkageError when an error occurs during linkage, and java.lang.ExceptionInInitializerError when an exception occurs during a class’s static initialization.</td>
</tr>
<tr>
<td><code>Annotation[] getAnnotations()</code></td>
<td>Returns an array (that’s possibly empty) containing all annotations that are declared for the class represented by this Class object.</td>
</tr>
<tr>
<td><code>Class&lt;?&gt;[] getClasses()</code></td>
<td>Returns an array containing Class objects representing all public classes and interfaces that are members of the class represented by this Class object. This includes public class and interface members inherited from superclasses and public class and interface members declared by the class. This method returns a zero-length array when this Class object has no public member classes or interfaces. This method also returns a zero-length array when this Class object represents a primitive type, an array class, or void.</td>
</tr>
<tr>
<td><code>Constructor[] getConstructors()</code></td>
<td>Returns an array containing java.lang.reflect.Constructor objects representing all the public constructors of the class represented by this Class object. A zero-length array is returned when the represented class has no public constructors, this Class object represents an array class, or this Class object represents a primitive type or void.</td>
</tr>
<tr>
<td><code>Annotation[] getDeclaredAnnotations()</code></td>
<td>Returns an array containing all annotations that are directly declared on the class represented by this Class object; inherited annotations are not included. The returned array might be empty.</td>
</tr>
<tr>
<td><code>Class&lt;?&gt;[] getDeclaredClasses()</code></td>
<td>Returns an array of Class objects representing all classes and interfaces declared as members of the class represented by this Class object. This includes public, protected, default (package) access, and private classes and interfaces. This method returns a zero-length array when the class declares no classes or interfaces as members, or when this Class object represents a primitive type, an array class, or void.</td>
</tr>
</tbody>
</table>

(continued)
Table 8-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor[] getDeclaredConstructors()</td>
<td>Returns an array of Constructor objects representing all constructors declared by the class represented by this Class object. These are public, protected, default (package) access, and private constructors. The returned array’s elements are not sorted and are not in any order. If the class has a default constructor, it’s included in the returned array. This method returns a zero-length array when this Class object represents an interface, a primitive type, an array class, or void.</td>
</tr>
<tr>
<td>Field[] getDeclaredFields()</td>
<td>Returns an array of java.lang.reflect.Field objects representing all fields declared by the class or interface represented by this Class object. This array includes public, protected, default (package) access, and private fields, but excludes inherited fields. The returned array’s elements are not sorted and are not in any order. This method returns a zero-length array when the class/interface declares no fields, or when this Class object represents a primitive type, an array class, or void.</td>
</tr>
<tr>
<td>Method[] getDeclaredMethods()</td>
<td>Returns an array of java.lang.reflect.Method objects representing all methods declared by the class or interface represented by this Class object. This array includes public, protected, default (package) access, and private methods, but excludes inherited methods. The elements in the returned array are not sorted and are not in any order. This method returns a zero-length array when the class or interface declares no methods, or when this Class object represents a primitive type, an array class, or void.</td>
</tr>
<tr>
<td>Field[] getFields()</td>
<td>Returns an array containing Field objects representing all public fields of the class or interface represented by this Class object, including those public fields inherited from superclasses and superinterfaces. The elements in the returned array are not sorted and are not in any order. This method returns a zero-length array when this Class object represents a class or interface that has no accessible public fields, or when this Class object represents an array class, a primitive type, or void.</td>
</tr>
<tr>
<td>Method[] getMethods()</td>
<td>Returns an array containing Method objects representing all public methods of the class or interface represented by this Class object, including those public methods inherited from superclasses and superinterfaces. Array classes return all the public member methods inherited from the java.lang.Object class. The elements in the returned array are not sorted and are not in any order. This method returns a zero-length array when this Class object represents a class or interface that has no public methods, or when this Class object represents a primitive type or void. The class initialization method &lt;clinit&gt;() (see Chapter 3) isn’t included in the returned array.</td>
</tr>
</tbody>
</table>
Table 8-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getModifiers()</td>
<td>Returns the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the virtual machine’s constants for public, protected, private, final, static, abstract, and interface; they should be decoded using the methods of class <code>java.lang.reflect.Modifier</code>. If the underlying class is an array class, its public, private, and protected modifiers are the same as those of its component type. If this Class object represents a primitive type or void, its public modifier is always true, and its protected and private modifiers are always false. If this Class object represents an array class, a primitive type, or void, its final modifier is always true and its interface modifier is always false. The values of its other modifiers are not determined by this specification.</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns the name of the class represented by this Class object.</td>
</tr>
<tr>
<td>Package getPackage()</td>
<td>Returns a <code>java.lang.Package</code> object that describes the package in which the class represented by this Class object is located, or returns null when the class is a member of the unnamed package.</td>
</tr>
<tr>
<td>Class&lt;? super T&gt; getSuperclass()</td>
<td>Returns the Class object representing the superclass of the entity (class, interface, primitive type, or void) represented by this Class object. When the Class object on which this method is called represents the Object class, an interface, a primitive type, or void, null is returned. When this object represents an array class, the Class object representing the Object class is returned.</td>
</tr>
<tr>
<td>boolean isAnnotation()</td>
<td>Returns true when this Class object represents an annotation type. If this method returns true, isInterface() also returns true because all annotation types are also interfaces.</td>
</tr>
<tr>
<td>boolean isEnum()</td>
<td>Returns true if and only if this class was declared as an enum in the source code.</td>
</tr>
<tr>
<td>boolean isInterface()</td>
<td>Returns true when this Class object represents an interface.</td>
</tr>
<tr>
<td>T newInstance()</td>
<td>Creates and returns a new instance of the class represented by this Class object. The class is instantiated as if by a <code>new</code> expression with an empty argument list. The class is initialized when it has not already been initialized. This method throws <code>java.lang.IllegalArgumentException</code> when the class or its noargument constructor isn’t accessible; <code>java.lang.InstantiationException</code> when this Class object represents an abstract class, an interface, an array class, a primitive type, or void, or when the class doesn’t have a noargument constructor (or when instantiation fails for some other reason); and <code>ExceptionInInitializerError</code> when initialization fails because the object threw an exception during initialization.</td>
</tr>
</tbody>
</table>
Table 8-1 identifies the Constructor, Field, Method, and Modifier members of the java.lang.reflect package. It also identifies other reflection-oriented types that belong to other packages. For example, Annotation (the superinterface of Override, SuppressWarnings, and all other annotation types) is a member of java.lang.annotation and Package is a member of java.lang.

### Obtaining a Class Object

Table 8-1’s description of the `forName()` method reveals one way to obtain a Class object. This method loads, links, and initializes a class or interface that isn’t in memory and returns a Class object representing the class or interface. Listing 8-3 demonstrates `forName()` and additional methods described in this table.

**Listing 8-3. Using Reflection to Decompile a Type**

```java
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class Decompiler
{
  public static void main(String[] args)
  {
    if (args.length != 1)
    {
      System.err.println("usage: java Decompiler classname");
      return;
    }
    try
    {
      decompileClass(Class.forName(args[0]), 0);
    }
    catch (ClassNotFoundException cnfe)
    {
      System.err.println("could not locate " + args[0]);
    }
  }

  static void decompileClass(Class<?> clazz, int indentLevel)
  {
    indent(indentLevel * 3);
    System.out.print(Modifier.toString(clazz.getModifiers()) + " ");
    if (clazz.isEnum())
      System.out.println("enum " + clazz.getName());
    else
      if (clazz.isInterface())
      {
        if (clazz.isAnnotation())
          System.out.print("@");
        System.out.println(clazz.getName());
        System.out.println(clazz.getName());
      }
  }
```

www.it-ebooks.info
else
    System.out.println(clazz);
indent(indentLevel * 3);
System.out.println("\n");
Field[] fields = clazz.getDeclaredFields();
for (int i = 0; i < fields.length; i++)
{
    indent(indentLevel * 3);
    System.out.println("   " + fields[i]);
}
Constructor[] constructors = clazz.getDeclaredConstructors();
if (constructors.length != 0 && fields.length != 0)
    System.out.println();
for (int i = 0; i < constructors.length; i++)
{
    indent(indentLevel * 3);
    System.out.println("   "+constructors[i]);
}
Method[] methods = clazz.getDeclaredMethods();
if (methods.length != 0 &&
    (fields.length != 0 || constructors.length != 0))
    System.out.println();
for (int i = 0; i < methods.length; i++)
{
    indent(indentLevel * 3);
    System.out.println("   "+methods[i]);
}
Method[] methodsAll = clazz.getMethods();
if (methodsAll.length != 0 &&
    (fields.length != 0 || constructors.length != 0 ||
     methods.length != 0 ||
     methodsAll.length != 0))
    System.out.println();
for (int i = 0; i < methodsAll.length; i++)
{
    indent(indentLevel * 3);
    System.out.println("   ALL PUBLIC METHODS");
    System.out.println();
}
if (methodsAll.length != 0)
{
    indent(indentLevel * 3);
    System.out.println("   ALL PUBLIC METHODS");
    System.out.println();
}
Listing 8-3 presents the source code to a decompiler tool that uses reflection to obtain information about the command-line argument, which must be a Java reference type (a class, for example). The decompiler outputs the type and name information for a class’s fields, constructors, methods, and nested types. It also outputs the members of interfaces, enums, and annotation types.

After verifying that one command-line argument has been passed to this application, main() calls forName() to return a Class object representing the class or interface identified by this argument.

forName() throws an instance of the checked ClassNotFoundException class when it cannot locate the class’s classfile (perhaps the classfile was erased before executing the application). It also throws LinkageError when a class’s classfile is malformed and ExceptionInInitializerError when a class’s static initialization fails.

Note ExceptionInInitializerError is often thrown as the result of a class initializer throwing an unchecked exception. For example, the class initializer in the following FailedInitialization class results in ExceptionInInitializerError because someMethod() throws java.lang.NullPointerException:

```java
public class FailedInitialization
{
    static
    {
        someMethod(null);
    }

    public static void someMethod(String s)
    {
        int len = s.length(); // s contains null
        System.out.println(s + "'s length is " + len + " characters");
    }
}
```
Assuming that forName() is successful, the returned object's reference is passed to decompileClass(), which decompiles the type. (decompileClass() is recursive in that it invokes itself for every encountered nested type.)

The decompileClass() method invokes Class's getModifiers(), isEnum(), getName(), isInterface(), isAnnotation(), getDeclaredFields(), getDeclaredConstructors(), getDeclaredMethods(), and getDeclaredClasses() methods to return different pieces of information about the loaded class or interface, which is subsequently output.

Much of the printing code focuses on making the output look nice as if it was a source code listing. The code manages indentation and only allows a newline character to be output to separate one section from another; a newline character isn't output unless content appears before and after the newline.

Compile Listing 8-3 (javac Decompiler.java) and run this application with java.lang.Byte as the solitary command-line argument (java Decompiler java.lang.Byte). You will observe the following output, which provides insight into how Byte is implemented:

```java
public final class java.lang.Byte
{
    public static final byte java.lang.Byte.MIN_VALUE
    public static final byte java.lang.Byte.MAX_VALUE
    public static final java.lang.Class java.lang.Byte.TYPE
    private final byte java.lang.Byte.value
    public static final int java.lang.Byte.SIZE
    private static final long java.lang.Byte.serialVersionUID
    public java.lang.Byte(byte)
    public java.lang.Byte(java.lang.String) throws java.lang.NumberFormatException
    public boolean java.lang.Byte.equals(java.lang.Object)
    public java.lang.String java.lang.Byte.toString()
    public static java.lang.String java.lang.Byte.toString(byte)
    public int java.lang.Byte.hashCode()
    public int java.lang.Byte.compareTo(java.lang.Byte)
    public int java.lang.Byte.compareTo(java.lang.Object)
    public byte java.lang.Byte.byteValue()
    public short java.lang.Byte.shortValue()
    public int java.lang.Byte.intValue()
    public long java.lang.Byte.longValue()
    public float java.lang.Byte.floatValue()
    public double java.lang.Byte.doubleValue()
    public static java.lang.Byte java.lang.Byte.valueOf(byte)
    NumberFormatException
    NumberFormatException
    public static int java.lang.Byte.compare(byte,byte)
    NumberFormatException
    NumberFormatException
    NumberFormatException
```

www.it-ebooks.info
ALL PUBLIC METHODS

public boolean java.lang.Byte.equals(java.lang.Object)
public java.lang.String java.lang.Byte.toString()
public static java.lang.String java.lang.Byte.toString(byte)
public int java.lang.Byte.hashCode()
public int java.lang.Byte.compareTo(java.lang.Byte)
public int java.lang.Byte.compareTo(java.lang.Object)
public byte java.lang.Byte.byteValue()
public short java.lang.Byte.shortValue()
public int java.lang.Byte.intValue()
public long java.lang.Byte.longValue()
public float java.lang.Byte.floatValue()
public double java.lang.Byte.doubleValue()
public static java.lang.Byte java.lang.Byte.valueOf(byte)
NumberFormatException
NumberFormatException
public static int java.lang.Byte.compare(byte,byte)
NumberFormatException
NumberFormatException
NumberFormatException

private static class java.lang.Byte$ByteCache
{
  static final java.lang.Byte[] java.lang.Byte$ByteCache.cache

  private java.lang.Byte$ByteCache()

  ALL PUBLIC METHODS

  public final void java.lang.Object.wait() throws java.lang.InterruptedException
  public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
  public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
  public boolean java.lang.Object.equals(java.lang.Object)
  public java.lang.String java.lang.Object.toString()
  public native int java.lang.Object.hashCode()
  public final native java.lang.Class java.lang.Object.getClass()
  public final native void java.lang.Object.notify()
  public final native void java.lang.Object.notifyAll()

  private static class java.lang.Byte$ByteCache
  {
    static final java.lang.Byte[] java.lang.Byte$ByteCache.cache

    private java.lang.Byte$ByteCache()

    ALL PUBLIC METHODS

    public final void java.lang.Object.wait() throws java.lang.InterruptedException
    public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
    public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
    public boolean java.lang.Object.equals(java.lang.Object)
    public java.lang.String java.lang.Object.toString()
    public native int java.lang.Object.hashCode()
    public final native java.lang.Class java.lang.Object.getClass()
    public final native void java.lang.Object.notify()
    public final native void java.lang.Object.notifyAll()
  }
}
Another way to obtain a Class object is to call Object’s getClass() method on an object reference; for example, Employee e = new Employee(); Class<? extends Employee> clazz = e.getClass(). The getClass() method doesn’t throw an exception because the class from which the object was created is already present in memory.

There is one more way to obtain a Class object, and that is to employ a class literal, which is an expression consisting of a class name, followed by a period separator that’s followed by reserved word class. Examples of class literals include Class<Employee> clazz = Employee.class; and Class<String> clazz = String.class.

Perhaps you’re wondering about how to choose between forName(), getClass(), and a class literal. To help you make your choice, the following list compares each competitor:

- forName() is very flexible in that you can dynamically specify any reference type by its package-qualified name. If the type isn’t in memory, it’s loaded, linked, and initialized. However, lack of compile-time type safety can lead to runtime failures.
- getClass() returns a Class object describing the type of its referenced object. When called on a superclass variable containing a subclass instance, a Class object representing the subclass type is returned. Because the class is in memory, type safety is assured.
- A class literal returns a Class object representing its specified class. Class literals are compact and the compiler enforces type safety by refusing to compile the source code when it cannot locate the literal’s specified class.

You can use class literals with primitive types, including void. Examples include int.class, double.class, and void.class. The returned Class object represents the class identified by a primitive type wrapper class’s TYPE field or java.lang.Void.TYPE. For example, each of int.class == java.lang.Integer.TYPE and void.class == Void.TYPE evaluates to true.

You can also use class literals with primitive type-based arrays. Examples include int[].class and double[].class. For these examples, the returned Class objects represent Class<int[]> and Class<double[]>.

**Instantiating a Dynamically Loaded Class**

One of Table 8-1’s methods not demonstrated in Listing 8-3 is newInstance(), which is useful for instantiating a dynamically loaded class provided that the class has a noargument constructor. The following code fragment demonstrates this:

```
try {
    Class<?> clazz = Class.forName("codecs.AVI");
    Codec codec = (Codec) clazz.newInstance();
    VideoPlayer player = new VideoPlayer(codec);
    player.play("movie.avi");
}
```
catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
}
catch (IllegalAccessException iae) {
    iae.printStackTrace();
}
catch (InstantiationException ie) {
    ie.printStackTrace();
}

This code fragment uses Class.forName() to attempt to load a hypothetical Audio Video Interleave (AVI) compressor/decompressor (codec), which is stored as AVI.class in the codecs package. If successful, the codec is instantiated via the newInstance() method. A hypothetical VideoPlayer class is instantiated and its instance is initialized to the codec, and the player is told to play the contents of movie.avi, which was encoded via this codec.

forName() throws ClassNotFoundException when it cannot find AVI.class. newInstance() throws IllegalAccessException when it cannot locate a noargument constructor in AVI.class and InstantiationException when instantiation fails (perhaps the classfile describes an interface or an abstract class).

**Constructor, Field, and Method**

Table 8-1’s method descriptions refer to Constructor, Field, and Method. Instances of these classes represent a class’s constructors and a class’s or an interface’s fields and methods.

Constructor represents a constructor and is generically declared as Constructor<T> where T identifies the class in which the constructor represented by Constructor is declared. Constructor declares various methods including the following methods:

- Annotation[] getDeclaredAnnotations() returns an array of all annotations declared on the constructor. The returned array has zero length when there are no annotations.
- Class<T> getDeclaringClass() returns a Class object that represents the class in which the constructor is declared.
- Class<?>[] getExceptionTypes() returns an array of Class objects representing the types of exceptions listed in the constructor’s throws clause. The returned array has zero length when there is no throws clause.
- String getName() returns the constructor’s name.
- Class<?>[] getParameterTypes() returns an array of Class objects representing the constructor’s parameter types, in declaration order. The returned array has zero length when the constructor declares no parameters.
Tip  If you want to instantiate a class via a constructor that takes arguments, you cannot use Class’s newInstance() method. Instead, you must use Constructor's T newInstance(Object... initargs) method to perform this task. Unlike Class's newInstance() method, which bypasses the compile-time exception checking that would otherwise be performed by the compiler, Constructor’s newInstance() method avoids this problem by wrapping any exception thrown by the constructor in an instance of the java.lang.reflect.InvocationTargetException class.

Field represents a field and declares various methods including the following getter methods:

- Object get(Object object) returns the value of the field for the specified object.
- boolean getBoolean(Object object) returns the value of the Boolean field for the specified object.
- byte getByte(Object object) returns the value of the byte integer field for the specified object.
- char getChar(Object object) returns the value of the character field for the specified object.
- double getDouble(Object object) returns the value of the double precision floating-point field for the specified object.
- float getFloat(Object object) returns the value of the floating-point field for the specified object.
- int getInt(Object object) returns the value of the integer field for the specified object.
- long getLong(Object object) returns the value of the long integer field for the specified object.
- short getShort(Object object) returns the value of the short integer field for the specified object.

get() returns the value of any type of field. In contrast, the other listed methods return the values of specific types of fields. These methods throw a NullPointerException instance when object is null and the field is an instance field, an IllegalArgumentException instance when object is not an instance of the class or interface declaring the underlying field (or not an instance of a subclass or interface implementor), and an IllegalAccessException instance when the underlying field cannot be accessed (perhaps the field is private).

Listing 8-4 demonstrates Field's getInt(Object) and getDouble(Object) methods along with their void setInt(Object obj, int i) and void setDouble(Object obj, double d) counterparts.
Listing 8-4. Reflectively Getting and Setting the Values of Instance and Class Fields

```java
import java.lang.reflect.Field;

class X {
    public int i = 10;
    public static final double PI = 3.14;
}

class FieldAccessDemo {
    public static void main(String[] args) {
        try {
            Class<?> clazz = Class.forName("X");
            X x = (X) clazz.newInstance();
            Field f = clazz.getField("i");
            System.out.println(f.getInt(x));       // Output: 10
            f.setInt(x, 20);
            System.out.println(f.getInt(x));       // Output: 20
            f = clazz.getField("PI");
            System.out.println(f.getDouble(null)); // Output: 3.14
            f.setDouble(x, 20);
            System.out.println(f.getDouble(null)); // Never executed
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Listing 8-4 declares classes X and FieldAccessDemo. I've included X's source code with FieldAccessDemo's source code for convenience. However, you can imagine this source code being stored in a separate source file.

FieldAccessDemo's main() method first attempts to load X and then tries to instantiate this class via newInstance(). If successful, the instance is assigned to reference variable x.

main() next invokes Class's Field getField(String name) method to return a Field instance that represents the public field identified by name, which happens to be i (in the first case) and PI (in the second case). This method throws java.lang.NoSuchFieldException when the named field doesn't exist.

Continuing, main() invokes Field's getInt() and setInt() methods (with an object reference) to get the instance field's initial value, change this value to another value, and get the new value. The initial and new values are output.

At this point, main() demonstrates class field access in a similar manner. However, it passes null to getInt() and setInt() because an object reference isn't required to access a class field. Because PI is declared final, the call to setInt() results in a thrown instance of the IllegalAccessException class.
**Note**  I've specified catch(Exception e) to avoid having to specify multiple catch blocks.

Method represents a method and declares various methods, including the following methods:

- **int getModifiers()** returns a 32-bit integer whose bit fields identify the method's reserved word modifiers (such as public, abstract, or static). These bit fields must be interpreted via the Modifier class. For example, you might specify (method.getModifiers() & Modifier.ABSTRACT) == Modifier.ABSTRACT to find out if the method (represented by the Method object whose reference is stored in method) is abstract—this expression evaluates to true when the method is abstract.

- **Class<?> getReturnType()** returns a Class object that represents the method's return type.

- **Object invoke(Object receiver, Object... args)** calls the method on the object identified by receiver (which is ignored when the method is a class method), passing the variable number of arguments identified by args to the called method. The invoke() method throws NullPointerException when receiver is null and the method being invoked is an instance method, IllegalAccessException when the method isn't accessible (perhaps it's private), IllegalArgumentException when an incorrect number of arguments are passed to the method (and other reasons), and InvocationTargetException when an exception is thrown from the called method.

- **boolean isVarArgs()** returns true when the method is declared to receive a variable number of arguments.

Listing 8-5 demonstrates Method's invoke(Object, Object...) method.

**Listing 8-5. Reflectively Invoking Instance and Class Methods**

```java
import java.lang.reflect.Method;

class X
{
    public void objectMethod(String arg)
    {
        System.out.println("Instance method: " + arg);
    }

    public static void classMethod()
    {
        System.out.println("Class method");
    }
}
```

www.it-ebooks.info
public class MethodInvocationDemo
{
    public static void main(String[] args)
    {
        try
        {
            Class<?> clazz = Class.forName("X");
            X x = (X) clazz.newInstance();
            Class[] argTypes = { String.class };
            Method method = clazz.getMethod("objectMethod", argTypes);
            Object[] data = { "Hello" };
            method.invoke(x, data); // Output: Instance method: Hello
            method = clazz.getMethod("classMethod", (Class<?>[]) null);
            method.invoke(null, (Object[]) null); // Output: Class method
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
    }
}

Listing 8-5 declares classes X and MethodInvocationDemo. MethodInvocationDemo's main() method first attempts to load X and then tries to instantiate this class via newInstance(). When successful, the instance is assigned to reference variable x.

main() next creates a one-element Class array that describes the types of objectMethod()'s parameter list. This array is used in the subsequent call to Class's Method getMethod(String name, Class<?>... parameterTypes) method to return a Method object for invoking a public method named objectMethod with this parameter list. This method throws java.lang.NoSuchMethodException when the named method doesn't exist.

Continuing, main() creates an Object array that specifies the data to be passed to the method's parameters; in this case, the array consists of a single String argument. It then reflectively invokes objectMethod() by passing this array along with the object reference stored in x to the invoke() method.

At this point, main() shows you how to reflectively invoke a class method. The (Class<?>[][]) and (Object[]) casts are used to suppress warning messages that have to do with variable numbers of arguments and null references. Notice that the first argument passed to invoke() is null when invoking a class method.

Accessible Objects

The java.lang.reflect.AccessibleObject class is the superclass of Constructor, Field, and Method. This superclass provides annotation-related methods; and methods for reporting a constructor's, field's, or method's accessibility (is it private?) and making an inaccessible constructor, field, or method accessible. AccessibleObject's methods include the following:

- <T extends Annotation> T getAnnotation(Class<T> annotationType) returns the constructor's, field's, or method's annotation of the specified type when such an annotation is present; otherwise, null returns.
- boolean isAccessible() returns true when the constructor, field, or method is accessible.
- boolean isAnnotationPresent(Class<? extends Annotation> annotationType) returns true when an annotation of the type specified by annotationType has been declared on the constructor, field, or method. This method takes inherited annotations into account.
- void setAccessible(boolean flag) attempts to make an inaccessible constructor, field, or method accessible when flag is true.

### Package

Table 8-1's method descriptions also referred to Package, a class that provides information about a package (see Chapter 5 for an introduction to packages). This information includes version details about the implementation and specification of a Java package, the name of the package, and an indication of whether or not the package has been sealed (all classes that are part of the package are archived in the same JAR file).

Table 8-2 describes some of Package’s methods.

**Table 8-2. Package Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getImplementationTitle()</td>
<td>Returns the title of this package's implementation, which might be null. The format of the title is unspecified.</td>
</tr>
<tr>
<td>String getImplementationVendor()</td>
<td>Returns the name of the vendor or organization that provides this package's implementation. This name might be null. The format of the name is unspecified.</td>
</tr>
<tr>
<td>String getImplementationVersion()</td>
<td>Returns the version number of this package's implementation, which might be null. This version string must be a sequence of positive decimal integers separated by periods and might have leading zeros.</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns the name of this package in standard dot notation; for example, java.lang.</td>
</tr>
<tr>
<td>static Package getPackage(String packageName)</td>
<td>Returns the Package object that is associated with the package identified as packageName or null when the package identified as packageName cannot be found. This method throws NullPointerException when packageName is null.</td>
</tr>
<tr>
<td>static Package[] getPackages()</td>
<td>Returns an array of all Package objects that are accessible to this method’s caller.</td>
</tr>
<tr>
<td>String getSpecificationTitle()</td>
<td>Returns the title of this package’s specification, which might be null. The format of the title is unspecified.</td>
</tr>
<tr>
<td>String getSpecificationVendor()</td>
<td>Returns the name of the vendor or organization that provides the specification that is implemented by this package. This name might be null. The format of the name is unspecified.</td>
</tr>
</tbody>
</table>
I have created a PackageInfo application that demonstrates most of Table 8-2's Package methods. Listing 8-6 presents this application's source code.

**Listing 8-6. Obtaining Information About a Package**

```java
public class PackageInfo {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("usage: java PackageInfo packageName [version] ");
            return;
        }
        Package pkg = Package.getPackage(args[0]);
        if (pkg == null) {
            System.err.println(args[0] + " not found");
            return;
        }
        System.out.println("Name: " + pkg.getName());
        System.out.println("Implementation title: " + pkg.getImplementationTitle());
        System.out.println("Implementation vendor: " + pkg.getImplementationVendor());
        System.out.println("Implementation version: " + pkg.getImplementationVersion());
        System.out.println("Specification title: " + pkg.getSpecificationTitle());
        System.out.println("Specification vendor: " + pkg.getSpecificationVendor());
    }
}
```

**Table 8-2. (continued)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String getSpecificationVersion()</code></td>
<td>Returns the version number of the specification of this package's implementation, which might be null. This version string must be a sequence of positive decimal integers separated by periods, and might have leading zeros.</td>
</tr>
<tr>
<td><code>boolean isCompatibleWith(String desired)</code></td>
<td>Checks this package to determine if it is compatible with the specified version string, by comparing this package's specification version with the desired version. Returns true when this package's specification version number is greater than or equal to the desired version number (this package is compatible); otherwise, returns false. This method throws NullPointerException when desired is null and java.lang.NumberFormatException when this package's version number or the desired version number is not in dotted form.</td>
</tr>
<tr>
<td><code>boolean isSealed()</code></td>
<td>Returns true when this package has been sealed; otherwise, returns false.</td>
</tr>
</tbody>
</table>
After compiling Listing 8-6 (javac PackageInfo.java), specify at least a package name on the command line. For example, java PackageInfo java.lang returns the following output under Java 7:

Name: java.lang
Implementation title: Java Runtime Environment
Implementation vendor: Oracle Corporation
Implementation version: 1.7.0.06
Specification title: Java Platform API Specification
Specification vendor: Oracle Corporation
Specification version: 1.7
Sealed: false

PackageInfo also lets you determine if the package’s specification is compatible with a specific version number. A package is compatible with its predecessors.

For example, java PackageInfo java.lang 1.6 outputs Compatible with 1.6: true, whereas java PackageInfo java.lang 1.8 outputs Compatible with 1.8: false.

You can also use PackageInfo with your own packages, which you learned to create in Chapter 5. For example, that chapter presented a logging package.

Copy PackageInfo.class into the directory containing the logging package directory (which contains the compiled classfiles), and execute the following command:

java PackageInfo logging

PackageInfo responds by displaying the following output:

logging not found

This error message is presented because getPackage() requires at least one classfile to be loaded from the package before it returns a Package object describing that package.

The only way to eliminate the previous error message is to load a class from the package. Accomplish this task by merging the following code fragment into Listing 8-6.

if (args.length == 3)
try
{
   Class.forName(args[2]);
}
catch (ClassNotFoundException cnfe) {
    System.err.println("cannot load " + args[2]);
    return;
}

This code fragment, which must precede

```java
Package pkg = Package.getPackage(args[0]);
```

loads the classfile named by the revised PackageInfo application's third command-line argument.

Run the new PackageInfo application via

```java
java PackageInfo logging 1.5 logging.File
```

and you will observe the following output, provided that File.class exists (you need to compile this package before specifying this command line)—this command line identifies logging's File class as the class to load:

```plaintext
Name: logging
Implementation title: null
Implementation vendor: null
Implementation version: null
Specification title: null
Specification vendor: null
Specification version: null
Sealed: false
```

It's not surprising to see all of these null values because no package information has been added to the logging package. Also, `NumberFormatException` is thrown from `isCompatibleWith()` because the logging package doesn't contain a specification version number in dotted form (it is null).

Perhaps the simplest way to place package information into the logging package is to create a `logging.jar` file in a similar manner to the example shown in Chapter 5. But first, you must create a small text file that contains the package information. You can choose any name for the file. Listing 8-7 reveals my choice of `manifest.mf`.

### Listing 8-7. `manifest.mf` Containing the Package Information

```
Implementation-Title: Logging Implementation
Implementation-Vendor: Jeff Friesen
Implementation-Version: 1.0a
Specification-Title: Logging Specification
Specification-Vendor: Jeff "JavaJeff" Friesen
Specification-Version: 1.0
Sealed: true
```

**Note** Make sure to press the Return/Enter key at the end of the final line (Sealed: true). Otherwise, you will probably observe Sealed: false in the output because this entry will not be stored in the logging package by the JDK's `jar` tool; `jar` is a bit quirky.
Execute the following command line to create a JAR file that includes logging and its files, and whose manifest, a special file named MANIFEST.MF that stores information about the contents of a JAR file, contains the contents of Listing 8-7:

```
jar cfm logging.jar manifest.mf logging
```

Alternatively, specify one of the following slightly longer command lines, which are equivalent to the former command line:

```
jar cfm logging.jar manifest.mf logging/*.class
jar cfm logging.jar manifest.mf logging\*.class
```

Either command line creates a JAR file named logging.jar (via the c [create] and f [file] options). It also merges the contents of manifest.mf (via the m [manifest] option) into MANIFEST.MF, which is stored in the package's/JAR's file META-INF directory.

**Note** To learn more about a JAR file's manifest, read the “JAR Manifest” section of the JDK documentation’s “JAR File Specification” page (http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#JAR_Manifest).

Assuming that the jar tool presents no error messages, execute the following Windows-oriented command line (or a command line suitable for your platform) to run PackageInfo and extract the package information from the logging package:

```
java -cp logging.jar;. PackageInfo logging 1.0 logging.File
```

The -cp command-line option lets you specify the classpath, which consists of logging.jar and the current directory (represented by the dot [. ] character). Fail to specify the dot and java outputs an error message complaining that it cannot locate PackageInfo.class.

This time, you should see the following output:

- Name: logging
- Implementation title: Logging Implementation
- Implementation vendor: Jeff Friesen (IV)
- Implementation version: 1.0a
- Specification title: Logging Specification
- Specification vendor: Jeff Friesen (SV)
- Specification version: 1.0
- Sealed: true
- Compatible with 1.0: true
Array

The java.lang.reflect package also includes an Array class whose class methods make it possible to reflectively create and access Java arrays. Listing 8-8 provides a demonstration.

Listing 8-8. Reflectively Creating and Accessing an Array

```java
import java.lang.reflect.Array;

public class ArrayDemo
{
    public static void main(String[] args)
    {
        String[] argsCopy = (String[]) Array.newInstance(String.class, args.length);
        for (int i = 0; i < args.length; i++)
            Array.set(argsCopy, i, args[i]);
        for (int i = 0; i < args.length; i++)
            System.out.println(Array.get(argsCopy, i));
    }
}
```

Listing 8-8 first invokes Array's Object newInstance(Class<?> componentType, int length) class method to create an array that can store String objects. It then copies all passed String arguments to this array, invoking Array's void set(Object array, int index, Object value) class method to store each object. Finally, it retrieves each stored object by invoking Array's Object get(Object array, int index) class method.

Compile Listing 8-8 (javac ArrayDemo.java) and run this application. For example, consider the following command:

```
java ArrayDemo a b c
```

This command generates the following output:

```
a
b
c
```

Exploring StringTokenizer

You'll occasionally want to extract a string's individual components. For example, you have the string "int x = 4;" and want to extract int, x, =, 4, and ; separately. Alternatively, you might have a comma-separated values (CSV) file where each line consists of multiple data items separated by commas and you want to read each value separately.

The task of breaking up a string into its individual components is known as parsing or tokenizing. The components themselves are known as tokens. (Technically, a token is a category, such as identifier, and the component is known as a lexeme, such as int.)
Java 1.0 introduced the `java.util.StringTokenizer` class to let applications tokenize strings. `StringTokenizer` lets you specify a string to be tokenized and a set of `delimiters` that separate successive tokens. This class declares three constructors for specifying these items:

- `StringTokenizer(String str)` constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is “\t\n\r\f”: the space character, the tab character, the newline character, the carriage-return character, and the form-feed character. Delimiter characters themselves won’t be treated as tokens. This constructor throws `NullPointerException` when you pass `null` to `str`.

- `StringTokenizer(String str, String delim)` constructs a string tokenizer for the specified string. The characters in the `delim` argument are the delimiters for separating tokens. Delimiter characters themselves won’t be treated as tokens. This constructor throws `NullPointerException` when you pass `null` to `delim`. Although it doesn’t throw an exception when you pass `null` to `delim`, trying to invoke other methods on the resulting `StringTokenizer` instance may result in `NullPointerException`.

- `StringTokenizer(String str, String delim, boolean returnDelims)` constructs a string tokenizer for the specified string. All characters in the `delim` argument are the delimiters for separating tokens. When the `returnDelims` flag is `true`, the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. When `returnDelims` is `false`, the delimiter characters are skipped and only serve as separators between tokens. This constructor throws `NullPointerException` when you pass `null` to `str`. Although it doesn’t throw an exception when you pass `null` to `delim`, invoking other methods on the resulting `StringTokenizer` instance may result in `NullPointerException`.

Additionally, `StringTokenizer` declares the following methods, which indicate that `StringTokenizer` instances maintain a current position for locating the next token:

- `int countTokens()` returns the number of times that this tokenizer’s `nextToken()` method can be called before it generates an exception. The current position is not advanced.

- `boolean hasMoreElements()` returns the same value as the `hasMoreTokens()` method. It exists because `StringTokenizer` implements the `java.util.Enumeration` interface (discussed in Chapter 9).

- `boolean hasMoreTokens()` returns `true` when more tokens are available from this tokenizer’s string; otherwise, it returns `false`. When this method returns `true`, a subsequent call to the noargument `nextToken()` method will successfully return a token.

- `Object nextElement()` returns the same value as the noargument `nextToken()` method, except that its return value is of type `Object` rather than `String`. It exists because `StringTokenizer` implements the `Enumeration` interface, and throws `java.util.NoSuchElementException` when there are no more tokens.

- `String nextToken()` returns the next token from this string tokenizer. It throws `NoSuchElementException` when there are no more tokens.
String nextToken(String delim) returns the next token from this string tokenizer. First, the set of characters considered to be delimiters by this StringTokenizer object is changed to be the characters specified by delim. Then, the next token in the string after the current position is returned. The current position is advanced beyond the recognized token. The new delimiter set remains the default after this call. This method throws NoSuchElementException when there are no more tokens and NullPointerException when you pass null to delim.

You would typically use StringTokenizer in a loop context, as follows:

```java
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens())
    System.out.println(st.nextToken());
```

This loop generates the following output:

this
is
a
test

Alternatively, you could specify the following loop context:

```java
StringTokenizer st = new StringTokenizer("this is a test");
Enumeration e = (Enumeration) st;
while (e.hasMoreElements())
    System.out.println(e.nextElement());
```

This loop generates the following output:

this
is
a
test

Of course, you would have to cast nextElement()’s Object return type to String before assigning the result to a String variable.

**Note**: StringTokenizer implements Enumeration so that you can create common code for enumerating tokens and legacy vector/hashtable (see Chapter 9) content.
Although you'll find StringTokenizer easy to use in your Android apps or non-Android Java programs, Java provides a more powerful alternative in the form of regular expressions (discussed in Chapter 13). To give you a taste for the power of regular expressions, you can easily bypass the previous loops for obtaining tokens by employing the following code fragment:

```java
String[] values = "this is a test".split("\\s");
```

The String class declares String[] split(String regex) and String[] split(String regex, int limit) methods that let you split a string into components that are separated by delimiters identified by the specified regular expression (regex). For example, `\s` represents the `\s` regular expression (backslashes must be doubled when placed in string literals), which stands for whitespace character. The string is split around whitespace character delimiters.

If you were to specify the following loop,

```java
for (int i = 0; i < values.length; i++)
    System.out.println(values[i]);
```

you would observe the following output:

```
this
is
a
test
```

### Exploring Timer and TimerTask

It's often necessary to schedule a task (a unit of work) for one-shot execution (the task runs only once) or for repeated execution at regular intervals. For example, you might schedule an alarm clock task to run only once (perhaps to wake you up in the morning) or schedule a nightly backup task to run at regular intervals. With either kind of task, you might want the task to run at a specific time in the future or after an initial delay.

You can use the Threads API (see Chapter 7) to accomplish task scheduling. However, Java 1.3 introduced a more convenient and simpler alternative in the form of java.util.Timer and java.util.TimerTask classes.

**Note** Android apps can use Timer and TimerTask but these classes must work with Android's android.os.Handler class or the android.app.Activity class's void runOnUiThread(Runnable action) method when the timer needs to update the user interface, which must occur on the user interface thread. For more information, check out stackoverflow's “Android timer? How?” topic (http://stackoverflow.com/questions/4597690/android-timer-how).
Timer provides a facility for scheduling TimerTasks for future execution on a background thread. Timer tasks may be scheduled for one-shot execution or for repeated execution at regular intervals. Before delving into the internals of these classes, check out Listing 8-9.

Listing 8-9. Displaying the Current Millisecond Value at Approximately 1-Second Intervals

```java
import java.util.Timer;
import java.util.TimerTask;

public class TimerDemo
{
    public static void main(String[] args)
    {
        TimerTask task = new TimerTask()
        {
            @Override
            public void run()
            {
                System.out.println(System.currentTimeMillis());
            }
        };
        Timer timer = new Timer();
timer.schedule(task, 0, 1000);
    }
}
```

Listing 8-9 describes an application that outputs the current time (in milliseconds) approximately every second. It first instantiates a TimerTask subclass (in this case, an anonymous class is used) whose overriding run() method outputs the time. It then instantiates Timer and invokes its schedule() method with this task as the first argument. The second and third arguments indicate that the task is scheduled for repeated execution after no initial delay and every 1000 milliseconds.

Compile Listing 8-9 (javac TimerDemo.java) and run this application (java TimerDemo). You should observe output that’s similar to the following truncated output:

```
1380933893664
1380933894666
1380933895668
1380933896668
1380933897670
1380933898672
```

**Timer in Depth**

Corresponding to each Timer object is a single background thread that’s used to execute all of the timer tasks sequentially. This thread is known as the task-execution thread.
Note  Timer scales to large numbers of concurrently scheduled timer tasks (thousands should present no problem). Internally, it uses a binary heap to represent its timer task queue so the cost to schedule a timer task is $O(\log n)$, where $n$ is the number of concurrently scheduled timer tasks. Check out Wikipedia’s “Big O notation” topic (http://en.wikipedia.org/wiki/Big_O_notation) to learn more about $O()$.

Timer declares the following constructors:

- **Timer()** creates a new timer whose task-execution thread doesn’t run as a daemon thread.

- **Timer(boolean isDaemon)** creates a new timer whose task-execution thread may be specified to run as a daemon (pass true to isDaemon). A daemon thread is called for scenarios where the timer will be used to schedule repeating “maintenance activities”, which must be performed for as long as the application is running, but shouldn’t prolong the application’s lifetime.

- **Timer(String name)** creates a new timer whose task-execution thread has the specified name. The task-execution thread doesn’t run as a daemon thread. This constructor throws NullPointerException when name is null.

- **Timer(String name, boolean isDaemon)** creates a new timer whose task-execution thread has the specified name and which may run as a daemon thread. This constructor throws NullPointerException when name is null.

Timer also declares the following methods:

- **void cancel()** terminates this timer, discarding any currently scheduled timer tasks. This method doesn’t interfere with a currently executing timer task (when it exists). After a timer has been terminated, its execution thread terminates gracefully and no more timer tasks may be scheduled on it. (Calling cancel() from within the run() method of a timer task that was invoked by this timer absolutely guarantees that the ongoing task execution is the last task execution that will ever be performed by this timer.) This method may be called repeatedly; the second and subsequent calls have no effect.

- **int purge()** removes all canceled timer tasks from this timer’s queue, and returns the number of timer tasks that have been removed. Calling purge() has no effect on the behavior of the timer, but eliminates references to the canceled timer tasks from the queue. When there are no external references to these timer tasks, they become eligible for garbage collection. (Most applications won’t need to call this method, which is designed for use by the rare application that cancels a large number of timer tasks. Calling purge() trades time for space: this method’s runtime may be proportional to $n + c \cdot \log n$, where $n$ is the number of timer tasks in the queue and $c$ is the number of canceled timer tasks.) It’s permissible to call purge() from within a timer task scheduled on this timer.

- **void schedule(TimerTask task, Date time)** schedules task for execution at time. When time is in the past, task is scheduled for immediate execution. This method throws IllegalArgumentException when time.getTime() is negative;
java.lang.IllegalStateException when task was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and NullPointerException when task or time is null. (You’ll learn about java.util. Date in Chapter 16.)

- void schedule(TimerTask task, Date firstTime, long period) schedules task for repeated fixed-delay execution, beginning at firstTime. Subsequent executions take place at approximately regular intervals, separated by period milliseconds. In **fixed-delay execution**, each execution is scheduled relative to the actual execution time of the previous execution. When an execution is delayed for any reason (such as garbage collection), subsequent executions are also delayed. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of period (assuming the system clock underlying Object.wait(long) is accurate). As a consequence, when the scheduled firstTime value is in the past, task is scheduled for immediate execution. Fixed-delay execution is appropriate for recurring tasks that require “smoothness.” In other words, this form of execution is appropriate for tasks where it’s more important to keep the frequency accurate in the short run than in the long run. This includes most animation tasks, such as blinking a cursor at regular intervals. It also includes tasks wherein regular activity is performed in response to human input, such as automatically repeating a character for as long as a key is held down. This method throws IllegalArgumentException when firstTime.getTime() is negative or period is negative or zero; IllegalStateException when task was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and NullPointerException when task or firstTime is null.

- void schedule(TimerTask task, long delay) schedules task for execution after delay milliseconds. This method throws IllegalArgumentException when delay is negative or delay + System.currentTimeMillis() is negative; IllegalStateException when task was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and NullPointerException when task is null.

- void schedule(TimerTask task, long delay, long period) schedules task for repeated fixed-delay execution, beginning after delay milliseconds. Subsequent executions take place at approximately regular intervals separated by period milliseconds. This method throws IllegalArgumentException when delay is negative, delay + System.currentTimeMillis() is negative, or period is negative or zero; IllegalStateException when task was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and NullPointerException when task is null.

- void scheduleAtFixedRate(TimerTask task, Date firstTime, long period) schedules task for repeated fixed-rate execution, beginning at time. Subsequent executions take place at approximately regular intervals, separated by period milliseconds. In **fixed-rate execution**, each execution is scheduled relative to the scheduled execution time of the initial execution. When an execution is delayed for any reason (such as garbage collection), two or more executions will occur in rapid succession to “catch up.” In the long run, the frequency of execution
will be exactly the reciprocal of *period* (assuming the system clock underlying `Object.wait(long)` is accurate). As a consequence, when the scheduled `firstTime` is in the past, any “missed” executions will be scheduled for immediate “catch up” execution. Fixed-rate execution is appropriate for recurring activities that are sensitive to absolute time (such as ringing a chime every hour on the hour, or running scheduled maintenance every day at a particular time). It’s also appropriate for recurring activities where the total time to perform a fixed number of executions is important, such as a countdown timer that ticks once every second for ten seconds. Finally, fixed-rate execution is appropriate for scheduling multiple repeating timer tasks that must remain synchronized with respect to one another. This method throws `IllegalArgumentException` when `firstTime.getTime()` is negative, or `period` is negative or zero; `IllegalStateException` when the task was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when the task or `firstTime` is null.

```java
void scheduleAtFixedRate(TimerTask task, long delay, long period)
```
schedules task for repeated fixed-rate execution, beginning after `delay` milliseconds. Subsequent executions take place at approximately regular intervals, separated by `period` milliseconds. This method throws `IllegalArgumentException` when `delay` is negative, `delay + System.currentTimeMillis()` is negative, or `period` is negative or zero; `IllegalStateException` when the task was already scheduled or canceled, the timer was canceled, or the task-execution thread terminated; and `NullPointerException` when the task is null.

After the last live reference to a `Timer` object goes away and all outstanding timer tasks have completed execution, the timer’s task-execution thread terminates gracefully (and becomes subject to garbage collection). However, this can take arbitrarily long to occur. (By default, the task-execution thread doesn’t run as a daemon thread, so it’s capable of preventing an application from terminating.) When an application wants to terminate a timer’s task-execution thread rapidly, the application should invoke `Timer`’s `cancel()` method.

When the timer’s task-execution thread terminates unexpectedly, for example, because its `stop()` method was invoked (you should never call any of `Thread`’s `stop()` methods because they’re inherently unsafe), any further attempt to schedule a timer task on the timer results in `IllegalStateException`, as if `Timer`’s `cancel()` method had been invoked.

**TimerTask in Depth**

Timer tasks are instances of classes that subclass the abstract `TimerTask` class, which implements the `Runnable` interface and offers the following methods:

```java
boolean cancel()
```
cancels this timer task. When the timer task has been scheduled for one-shot execution and hasn’t yet run or when it hasn’t yet been scheduled, it will never run. When the timer task has been scheduled for repeated execution, it will never run again. (When the timer task is running when this call occurs, the timer task will run to completion, but will never run again.) Calling `cancel()` from within the `run()` method of a repeating timer task.

www.it-ebooks.info
absolutely guarantees that the timer task won’t run again. This method may be called repeatedly; the second and subsequent calls have no effect. This method returns true when this timer task is scheduled for one-shot execution and hasn’t yet run or when this timer task is scheduled for repeated execution. It returns false when the timer task was scheduled for one-shot execution and has already run, when the timer task was never scheduled, or when the timer task was already canceled. (Loosely speaking, this method returns true when it prevents one or more scheduled executions from taking place.)

- **void run()** provides the timer task’s code. You must override this abstract method to perform a useful activity.

- **long scheduledExecutionTime()** returns the scheduled execution time of the most recent actual execution of this timer task. (When this method is invoked while timer task execution is in progress, the return value is the scheduled execution time of the ongoing timer task execution.) This method is typically invoked from within a task’s run() method to determine whether the current execution of the timer task is sufficiently timely to warrant performing the scheduled activity. For example, you would specify code similar to if (System.currentTimeMillis() - scheduledExecutionTime() >= MAX_TARDINESS) return; at the start of the run() method to abort the current timer task execution when it’s not timely. This method is typically not used in conjunction with fixed-delay execution repeating timer tasks because their scheduled execution times are allowed to drift over time and are thus not terribly significant. scheduledExecutionTime() returns the time at which the most recent execution of this timer task was scheduled to occur, in the format returned by Date.getTime(). The return value is undefined when the timer task has yet to commence its first execution.

Timer tasks should complete quickly. When a timer task takes too long to complete, it “hogs” the timer’s task-execution thread, delaying the execution of subsequent timer tasks, which may “bunch up” and execute in rapid succession if and when the offending timer task finally completes.

### One-Shot Execution and Repeated Execution with Cancellation

Listing 8-9 demonstrates a timer executing a timer task indefinitely. You can also execute a timer task exactly once or execute it repeatedly until you want to cancel the task. I’ve created a simple TimerDemo application that demonstrates Timer and TimerTask in one-shot execution and repeated execution with cancellation contexts. Check out Listing 8-10.

**Listing 8-10. Demonstrating One-Shot Execution and Repeated Execution with Cancellation**

```java
import java.util.Timer;
import java.util.TimerTask;

public class TimerDemo
{
    public static void main(String[] args)
    {
        Timer t = new Timer(true);
```
t.schedule(new TimerTask()
{
    @Override
    public void run()
    {
        System.out.println("one-shot timer task executing");
    }
}, 2000); // Execute one-shot timer task after 2-second delay.
System.out.println("main thread is sleeping for 4 seconds");
try { Thread.sleep(4000); } catch (InterruptedException ie) {}
System.out.println("main thread has woken up");
t = new Timer();
t.schedule(new TimerTask()
{
    int i; // initializes to 0 by default
    @Override
    public void run()
    {
        System.out.println("repeated timer task is running");
        if (++i == 6)
        {
            System.out.println("canceling repeated timer task");
            cancel();
            System.out.println("canceled");
            System.exit(0);
        }
    }
}, 2000, 500);
System.out.println("main thread is terminating");
}

Listing 8-10’s main thread first creates a timer whose task-execution thread is daemon, and then schedules a one-shot timer task to execute two seconds later. The main thread then sleeps for four seconds to give you an opportunity to observe the timer task’s execution via an output message.

After the main thread awakens, it creates a second timer whose task-execution thread isn’t daemon, and then schedules a repeating timer task to run two seconds later and repeat every half-second. The main thread then terminates.

Because the second timer’s task-execution thread isn’t daemon, the application doesn’t terminate. Instead, the timer repeatedly executes its timer task six times before the final timer task execution cancels itself and invokes System.exit(0); to terminate the application. Without System.exit(0);, the application would never end because the timer’s non-daemon task-execution thread is still running and waiting to execute subsequently scheduled timer tasks.

Compile Listing 8-10 (javac TimerDemo.java) and run this application (java TimerDemo). You should observe the following output:

main thread is sleeping for 4 seconds
one-shot timer task executing
main thread has woken up

www.it-ebooks.info
main thread is terminating
repeated timer task is running
repeated timer task is running
repeated timer task is running
repeated timer task is running
repeated timer task is running
repeated timer task is running
canceling repeated timer task
canceled

EXERCISES

The following exercises are designed to test your understanding of Chapter 8’s content.

1. What does the Random class accomplish?
2. Define root set of references.
3. True or false: The garbage collector can collect reachable and unreachable objects.
4. Identify the different levels of reachability.
5. What is the different between a soft reference and a weak reference?
6. Identify the classes that comprise the References API.
7. Which reference class would you use to implement caches of objects that are expensive timewise to create?
8. Which reference class would you use as a replacement for the finalize() method?
9. What are some of the capabilities offered by the Reflection API?
10. Reflection should not be used indiscriminately. Why not?
11. Identify the class that is the entry point into the Reflection API.
12. True or false: All of the Reflection API is contained in the java.lang.reflect package?
13. What are the three ways to obtain a Class object?
14. True or false: You can use class literals with primitive types.
15. How do you instantiate a dynamically loaded class?
16. What method do you invoke to obtain a constructor’s parameter types?
17. What does Class’s Field getField(String name) method do when it cannot locate the named field?
18. How do you determine if a method is declared to receive a variable number of arguments?
19. True or false: You can reflectively make a private method accessible.
20. What is the purpose of Package’s isSealed() method?
21. True or false: getPackage() requires at least one classfile to be loaded from the package before it returns a Package object describing that package.
22. How do you reflectively create and access a Java array?

23. What is the purpose of the StringTokenizer class?

24. Identify the standard class library’s convenient and simpler alternative to the Threads API for scheduling task execution.

25. True or false: Timer() creates a new timer whose task-execution thread runs as a daemon thread.

26. Define fixed-delay execution.

27. Which methods do you call to schedule a task for fixed-delay execution?

28. Define fixed-rate execution.

29. What is the difference between Timer's cancel() method and TimerTask's cancel() method?

30. Create a Guess application that uses Random to generate a random integer from 0 to 25. Then, the application repeatedly asks the user to guess which integer was chosen by entering a letter from a through z. The application continues by comparing the entered letter with the random integer (which is offset by lowercase letter a) to learn if the user’s choice was too low, too high, or just right; and outputs an appropriate message.

31. Create a Classify application that uses Class's forName() method to load its single command-line argument, which will represent a package-qualified annotation type, enum, interface, or class (the default). Use a chained if-else statement along with the appropriate Class methods and System.out.println() to identify the type and output Annotation, Enum, Interface, or Class.

32. Create a Tokenize application that uses StringTokenizer to extract the month, day, year, hour, minute, and second fields from the string 03-12-2014 03:05:20, which are then output.

33. Create a BackAndForth application that uses Timer and TimerTask to repeatedly move an asterisk forward 20 steps and then backward 20 steps. The asterisk is output via System.out.println().

Summary

The Math class's random() method is implemented in terms of the Random class, whose instances are known as random number generators. Random generates a sequence of random numbers by starting with a special 48-bit seed. This value is subsequently modified via a mathematical algorithm that is known as a linear congruential generator.

Random declares a pair of constructors, a setSeed() method for setting the random number generator’s seed, and several “next”-prefixed methods that return the next number in the sequence as a Boolean value, an integer, a long integer, a floating-point value, or a double precision floating-point value. There is even a method for generating a sequence of random bytes.

When an application runs, its execution reveals a root set of references, which changes during the application’s execution. Garbage collectors use the root set to determine if an object is reachable or unreachable. The garbage collector cannot collect reachable objects. Instead, it can only collect objects that, starting from the root set of references, cannot be reached.
Reachable objects are classified as strongly reachable, softly reachable, weakly reachable, and phantom reachable. Unlike strongly reachable objects, softly, weakly, and phantom reachable objects can be garbage collected. References to softly, weakly, and phantom reachable objects are encapsulated by SoftReference, WeakReference, and PhantomReference objects.

SoftReference, WeakReference, and PhantomReference extend the Reference class. When you associate a ReferenceQueue instance with one of these Reference subclass objects, the Reference subclass object is added to the queue when the object to which its encapsulated reference refers (this object is known as a referent) becomes garbage.

Java’s Reflection API offers a third RTTI form in which applications can dynamically load and learn about loaded classes and other reference types. The API also lets applications instantiate classes, call methods, access fields, and perform other tasks reflectively. This form of RTTI is known as reflection or introspection.

The java.lang package’s Class class is the entry point into the Reflection API, whose types are stored mainly in the java.lang.reflect package. You can obtain a Class object by invoking Class’s forName() method, by invoking Object’s getClass() method, or by using a class literal, which is the name of a class followed by a .class suffix.

The java.lang.reflect package declares Constructor, Field, and Method classes that represent constructors, fields, and methods. Each of these classes extends the AccessibleObject class, which provides methods for determining if the constructor, field, or method is accessible; and for changing the constructor’s, field’s, or method’s accessibility.

The Package class provides access to package information. This information includes version information about the implementation and specification of a Java package, the package’s name, and an indication of whether the package is sealed or not. Also, the Array class provides class methods that make it possible to reflectively create and access Java arrays.

You’ll occasionally want to extract a string’s individual components. The task of breaking up a string into its individual components is known as parsing or tokenizing. The components themselves are known as tokens. Java 1.0 introduced the StringTokenizer class to let applications tokenize strings.

StringTokenizer provides constructors for specifying the string to be tokenized and the set of delimiters that separate successive tokens. The number of tokens can be obtained by invoking the countTokens() method.

The hasMoreElements() and hasMoreTokens() methods tell you whether there are more tokens to extract. The nextToken() and nextElement() methods return the next token. One of the nextToken() methods also lets you specify a new set of delimiters.

It’s often necessary to schedule a task for one-shot execution in which the task runs only once or for repeated execution at regular intervals. You can use the Threads API to accomplish task scheduling. However, Java 1.3 introduced a more convenient and simpler alternative in the form of Timer and TimerTask classes.

Timer provides a facility for scheduling TimerTasks for future execution on a background thread. Timer tasks may be scheduled for one-shot execution or for repeated execution at regular intervals. Corresponding to each Timer object is a single background thread that’s used to execute all of the timer tasks sequentially.

This chapter completes my tour of Java’s basic APIs. Chapter 9 explores Java’s Collections Framework and classic collections APIs.
Applications often must manage collections of objects. Although you can use arrays for this purpose, they are not always a good choice. For example, arrays have fixed sizes, making it tricky to determine an optimal size when you need to store a variable number of objects. Also, arrays can be indexed by integers only, making them unsuitable for mapping arbitrary objects to other objects.

The standard class library provides the Collections Framework and legacy utility APIs to manage collections on behalf of applications. In this chapter, I first present this framework and then introduce you to these legacy APIs (in case you encounter them in legacy code). As you will discover, some of the legacy APIs are still useful.

Note  Java’s Concurrency Utilities API suite (discussed in Chapter 10) extends the Collections Framework.

Exploring Collections Framework Fundamentals

The Collections Framework is a group of types (mainly located in the java.util package) that offers a standard architecture for representing and manipulating collections, which are groups of objects stored in instances of classes designed for this purpose. This framework’s architecture is divided into three sections:

- **Core interfaces**: The framework provides core interfaces for manipulating collections independently of their implementations.
- **Implementation classes**: The framework provides classes that offer different core interface implementations to address performance and other requirements.
- **Utility classes**: The framework provides utility classes with methods for sorting arrays, obtaining synchronized collections, and more.
The core interfaces include `java.lang.Iterable`, `Collection`, `List`, `Set`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `Map`, `SortedMap`, and `NavigableMap`. `Collection` extends `Iterable`; `List`, `Set`, and `Queue` each extend `Collection`; `SortedSet` extends `Set`; `NavigableSet` extends `SortedSet`; `Deque` extends `Queue`; `SortedMap` extends `Map`; and `NavigableMap` extends `SortedMap`.

Figure 9-1 illustrates the core interfaces hierarchy (arrows point to parent interfaces).

![Diagram of the Collections Framework hierarchy](image)

**Figure 9-1. The Collections Framework is based on a hierarchy of core interfaces**

The framework's implementation classes include `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, `LinkedHashSet`, `EnumSet`, `PriorityQueue`, `ArrayDeque`, `TreeMap`, `HashMap`, `LinkedHashMap`, `IdentityHashMap`, `WeakHashMap`, and `EnumMap`. The name of each concrete class ends in a core interface name, identifying the core interface on which it is based.

**Note** Additional implementation classes are part of the concurrency utilities.

The framework's implementation classes also include the abstract `AbstractCollection`, `AbstractList`, `AbstractSequentialList`, `AbstractSet`, `AbstractQueue`, and `AbstractMap` classes. These classes offer skeletal implementations of the core interfaces to facilitate the creation of concrete implementation classes.

Finally, the framework provides two utility classes: `Arrays` and `Collections`.

**Comparable vs. Comparator**

A collection implementation stores its elements in some order (arrangement). This order may be unsorted, or it may be sorted according to some criterion (such as alphabetical, numerical, or chronological).

A sorted collection implementation defaults to storing its elements according to their natural ordering. For example, the natural ordering of `java.lang.String` objects is lexicographic or dictionary (also known as alphabetical) order.
A collection cannot rely on `equals()` to dictate natural ordering because this method can only determine if two elements are equivalent. Instead, element classes must implement the `java.lang.Comparable<T>` interface and its `int compareTo(T o)` method.

**Note** According to `Comparable`'s Oracle-based Java documentation, this interface is considered to be part of the Collections Framework even though it is a member of the `java.lang` package.

A sorted collection uses `compareTo()` to determine the natural ordering of this method’s element argument `o` in a collection. `compareTo()` compares argument `o` with the current element (which is the element on which `compareTo()` was called) and does the following:

- It returns a negative value when the current element should precede `o`.
- It returns a zero value when the current element and `o` are the same.
- It returns a positive value when the current element should succeed `o`.

When you need to implement `Comparable`'s `compareTo()` method, there are some rules that you must follow. The following rules are similar to those shown in Chapter 4 for implementing the `equals()` method:

- `compareTo()` **must be reflexive**: For any nonnull reference value `x`, `x.compareTo(x)` must return 0.
- `compareTo()` **must be symmetric**: For any nonnull reference values `x` and `y`, `x.compareTo(y) == -y.compareTo(x)` must hold.
- `compareTo()` **must be transitive**: For any nonnull reference values `x`, `y`, and `z`, if `x.compareTo(y) > 0` is true, and if `y.compareTo(z) > 0` is true, then `x.compareTo(z) > 0` must also be true.

Also, `compareTo()` should throw `java.lang.NullPointerException` when the null reference is passed to this method. However, you don’t need to check for null because this method throws `NullPointerException` when it attempts to access a null reference’s nonexistent members.

**Note** Before Java 5 and its introduction of generics, `compareTo()`’s argument was of type `java.lang.Object` and had to be cast to the appropriate type before the comparison could be made. The cast operator would throw a `java.lang.ClassCastException` instance when the argument’s type was not compatible with the cast.

You might occasionally need to store in a collection objects that are sorted in some order that differs from their natural ordering. In this case, you would supply a comparator to provide that ordering.
A comparator is an object whose class implements the Comparator interface. This interface, whose
generic type is Comparator<T>, provides the following pair of methods:

- int compare(T o1, T o2) compares both arguments for order. This method returns 0 when o1 equals o2, a negative value when o1 is less than o2, and a positive value when o1 is greater than o2.

- boolean equals(Object o) returns true when o “equals” this Comparator in that o is also a Comparator and imposes the same ordering. Otherwise, this method returns false.

Note  Comparator declares equals() because this interface places an extra condition on this method’s contract. Additionally, this method can return true only if the specified object is also a comparator and it imposes the same ordering as this comparator. You don’t have to override equals(), but doing so may improve performance by allowing programs to determine that two distinct comparators impose the same order.

Chapter 6 provided an example that illustrated implementing Comparable, and you will discover another example later in this chapter. Also, in this chapter, I will present examples of implementing Comparator.

**Iterable and Collection**

Most of the core interfaces are rooted in Iterable and its Collection subinterface. Their generic
types are Iterable<T> and Collection<E>.

Iterable describes any object that can return its contained objects in some sequence. This interface declares an Iterator<T> iterator() method that returns an Iterator instance for iterating over all of the contained objects.

Collection represents a collection of objects that are known as elements. This interface provides methods that are common to the Collection subinterfaces on which many collections are based. Table 9-1 describes these methods.
### Table 9-1. Collection Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add(E e)</td>
<td>Adds element e to this collection. Returns true if this collection was modified as a result; otherwise, returns false. (Attempting to add e to a collection that doesn’t permit duplicates and already contains a same-valued element results in e not being added.) This method throws java.lang.UnsupportedOperationException when add() is not supported, ClassCastException when e’s class is not appropriate for this collection, java.lang.IllegalArgumentException when some property of e prevents it from being added to this collection, NullPointerException when e contains the null reference and this collection doesn’t support null elements, and java.lang.IllegalArgumentException when the element cannot be added at this time because of insertion restrictions. IllegalStateException signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java/Android environment or application is not in an appropriate state for the requested operation. It is often thrown when you try to add an element to a bounded queue (a queue with a maximum length) and the queue is full.</td>
</tr>
<tr>
<td>boolean addAll(Collection&lt;? extends E&gt; c)</td>
<td>Adds all elements of collection c to this collection. Returns true if this collection was modified as a result; otherwise, returns false. This method throws UnsupportedOperationException when this collection doesn’t support addAll(), ClassCastException when the class of one of c’s elements is inappropriate for this collection, IllegalArgumentException when some property of an element prevents it from being added to this collection, NoSuchElementException when c contains the null reference or when one of its elements is null and this collection doesn’t support null elements, and IllegalStateException when not all the elements can be added at this time because of insertion restrictions.</td>
</tr>
<tr>
<td>void clear()</td>
<td>Removes all elements from this collection. This method throws UnsupportedOperationException when this collection doesn’t support clear().</td>
</tr>
<tr>
<td>boolean contains(Object o)</td>
<td>Returns true when this collection contains o; otherwise, returns false. This method throws ClassCastException when the class of o is inappropriate for this collection and NullPointerException when o contains the null reference and this collection doesn’t support null elements.</td>
</tr>
<tr>
<td>boolean containsAll(Collection&lt;?&gt; c)</td>
<td>Returns true when this collection contains all of the elements that are contained in the collection specified by c; otherwise, returns false. This method throws ClassCastException when the class of one of c’s elements is inappropriate for this collection and NullPointerException when c contains the null reference or when one of its elements is null and this collection doesn’t support null elements.</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object o)</td>
<td>Compares o with this collection and returns true when o equals this collection; otherwise, returns false.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns this collection’s hash code. Equal collections have equal hash codes.</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns true when this collection contains no elements; otherwise, returns false.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; iterator()</td>
<td>Returns an Iterator instance for iterating over all of the elements contained in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee). This Iterable method is redeclared in Collection for convenience.</td>
</tr>
<tr>
<td>boolean remove(Object o)</td>
<td>Removes the element identified as o from this collection. Returns true when the element is removed; otherwise, returns false. This method throws UnsupportedOperationException when this collection doesn’t support remove(), ClassCastException when the class of o is inappropriate for this collection, and NullPointerException when o contains the null reference and this collection doesn’t support null elements.</td>
</tr>
<tr>
<td>boolean removeAll(Collection&lt;?&gt; c)</td>
<td>Removes all of the elements from this collection that are also contained in collection c. Returns true when this collection is modified by this operation; otherwise, returns false. This method throws UnsupportedOperationException when this collection doesn’t support removeAll(), ClassCastException when the class of one of c’s elements is inappropriate for this collection, and NullPointerException when c contains the null reference or when one of its elements is null and this collection doesn’t support null elements.</td>
</tr>
<tr>
<td>boolean retainAll(Collection&lt;?&gt; c)</td>
<td>Retains all of the elements in this collection that are also contained in collection c. Returns true when this collection is modified by this operation; otherwise, returns false. This method throws UnsupportedOperationException when this collection doesn’t support retainAll(), ClassCastException when the class of one of c’s elements is inappropriate for this collection, and NullPointerException when c contains the null reference or when one of its elements is null and this collection doesn’t support null elements.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of elements contained in this collection, or java.lang.Integer.MAX_VALUE when there are more than Integer.MAX_VALUE elements contained in the collection.</td>
</tr>
</tbody>
</table>
Table 9-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object[] toArray()</td>
<td>Returns an array containing all of the elements stored in this collection. If this collection makes any guarantees as to what order its elements are returned in by its iterator, this method returns the elements in the same order. The returned array is “safe” in that no references to it are maintained by this collection. (In other words, this method allocates a new array even when this collection is backed by an array.) The caller can safely modify the returned array.</td>
</tr>
<tr>
<td>&lt;T&gt; T[] toArray(T[] a)</td>
<td>Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it’s returned in the array. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection. This method throws NullPointerException when null is passed to a, and java.lang.ArrayStoreException when a’s runtime type is not a supertype of the runtime type of every element in this collection.</td>
</tr>
</tbody>
</table>

Table 9-1 reveals three exceptional things about various Collection methods. First, some methods can throw instances of the UnsupportedOperationException class. For example, add() throws UnsupportedOperationException when you attempt to add an object to an immutable (unmodifiable) collection (discussed later in this chapter).

Second, some of Collection’s methods can throw instances of the ClassCastException class. For example, remove() throws ClassCastException when you attempt to remove an entry (also known as mapping) from a tree-based map whose keys are Strings, but specify a non-String key instead.

Finally, Collection’s add() and addAll() methods throw IllegalArgumentException instances when some property (attribute) of the element to be added prevents it from being added to this collection. For example, a third-party collection class’s add() and addAll() methods might throw this exception when they detect negative Integer values.

Note Perhaps you’re wondering why remove() is declared to accept any Object argument instead of accepting only objects whose types are those of the collection. In other words, why is remove() not declared as boolean remove(E e)? Also, why are containsAll(), removeAll(), and retainAll() not declared with an argument of type Collection<? extends E> to ensure that the collection argument only contains elements of the same type as the collection on which these methods are called? The answer to these questions is the need to maintain backward compatibility. The Collections Framework was introduced before Java 5 and its introduction of generics. To let legacy code written before version 5 continue to compile, these four methods were declared with weaker type constraints.
CHAPTER 9: Exploring the Collections Framework

Iterator and the Enhanced For Loop Statement

By extending Iterable, Collection inherits that interface's iterator() method, which makes it possible to iterate over a collection. iterator() returns an instance of a class that implements the Iterator interface, whose generic type is expressed as Iterator<E> and which declares the following three methods:

- boolean hasNext() returns true when this Iterator instance has more elements to return; otherwise, this method returns false.
- E next() returns the next element from the collection associated with this Iterator instance, or throws NoSuchElementException when there are no more elements to return.
- void remove() removes the last element returned by next() from the collection associated with this Iterator instance. This method can be called only once per next() call. The behavior of an Iterator instance is unspecified when the underlying collection is modified while iteration is in progress in any way other than by calling remove(). This method throws UnsupportedOperationException when it is not supported by this Iterator, and IllegalStateException when remove() has been called without a previous call to next() or when multiple remove() calls occur with no intervening next() calls.

The following example shows you how to iterate over a collection after calling iterator() to return an Iterator instance:

```java
Collection<String> col = ... // This code doesn't compile because of the ...
// Add elements to col.
Iterator iter = col.iterator();
while (iter.hasNext())
    System.out.println(iter.next());
```

The while loop repeatedly calls the iterator’s hasNext() method to determine whether or not iteration should continue, and (if it should continue) the next() method to return the next element from the associated collection.

Because this idiom is commonly used, Java 5 introduced syntactic sugar to the for loop statement to simplify iteration in terms of the idiom. This sugar makes this statement appear like the foreach statement found in languages such as Perl and is revealed in the following simplified equivalent of the previous example:

```java
Collection<String> col = ... // This code doesn't compile because of the ...
// Add elements to col.
for (String s: col)
    System.out.println(s);
```

This sugar hides col.iterator(), a method call that returns an Iterator instance for iterating over col's elements. It also hides calls to Iterator's hasNext() and next() methods on this instance. You interpret this sugar to read as follows: “for each String object in col, assign this object to s at the start of the loop iteration.”
CHAPTER 9: Exploring the Collections Framework

Note The enhanced for loop statement is also useful in an arrays context, in which it hides the array index variable. Consider the following example:

```java
String[] verbs = { "run", "walk", "jump" };
for (String verb: verbs)
    System.out.println(verb);
```

This example, which reads as “for each String object in the verbs array, assign that object to verb at the start of the loop iteration,” is equivalent to the following example:

```java
String[] verbs = { "run", "walk", "jump" };
for (int i = 0; i < verbs.length; i++)
    System.out.println(verbs[i]);
```

The enhanced for loop statement is limited in that you cannot use this statement where access to the iterator is required to remove an element from a collection. Also, it’s not usable where you must replace elements in a collection/array during a traversal, and it cannot be used where you must iterate over multiple collections or arrays in parallel.

### Autoboxing and Unboxing

Developers who believe that Java should support only reference types have complained about Java’s support for primitive types. One area where the dichotomy of Java’s type system is clearly seen is the Collections Framework: you can store objects but not primitive-type-based values in collections.

Although you cannot directly store a primitive-type-based value in a collection, you can indirectly store this value by first wrapping it in an object created from a primitive type wrapper class such as `Integer` and then storing this primitive type wrapper class instance in the collection—see the following example:

```java
Collection<Integer> col = ...; // This code doesn't compile because of the ...
int x = 27;
col.add(new Integer(x));       // Indirectly store int value 27 via an Integer object.
```

The reverse situation is also tedious. When you want to retrieve the int from `col`, you must invoke `Integer`'s `intValue()` method (which, if you recall, is inherited from `Integer`’s `java.lang.Number` superclass). Continuing on from this example, you would specify `int y = col.iterator().next().intValue();` to assign the stored 32-bit integer to `y`.

To alleviate this tedium, Java 5 introduced autoboxing and unboxing, a pair of complementary syntactic sugar-based language features that make primitive-type values appear more like objects. (This “sleight of hand” isn’t complete because you cannot specify expressions such as `27.doubleValue()`.)
Autoboxing automatically boxes (wraps) a primitive-type value in an object of the appropriate primitive type wrapper class whenever a primitive-type value is specified but a reference is required. For example, you could change the example's third line to `col.add(x)`; and have the compiler box x into an Integer object.

Unboxing automatically unboxes (unwraps) a primitive-type value from its wrapper object whenever a reference is specified but a primitive-type value is required. For example, you could specify `int y = col.iterator().next();` and have the compiler unbox the returned Integer object to int value 27 prior to the assignment.

Although autoboxing and unboxing were introduced to simplify working with primitive-type values in a collections context, these language features can be used in other contexts; and this arbitrary use can lead to a problem that is difficult to understand without knowledge of what is happening behind the scenes. Consider the following example:

```java
Integer i1 = 127;
Integer i2 = 127;
System.out.println(i1 == i2); // Output: true
System.out.println(i1 < i2); // Output: false
System.out.println(i1 > i2); // Output: false
System.out.println(i1 + i2); // Output: 254
i1 = 30000;
i2 = 30000;
System.out.println(i1 == i2); // Output: false
System.out.println(i1 < i2); // Output: false
System.out.println(i1 > i2); // Output: false
i2 = 30001;
System.out.println(i1 < i2); // Output: true
System.out.println(i1 + i2); // Output: 60001
```

With one exception, this example's output is as expected. The exception is the `i1 == i2` comparison where each of `i1` and `i2` contains 30000. Instead of returning true, as is the case where each of `i1` and `i2` contains 127, `i1 == i2` returns false. What is causing this problem?

Examine the generated code and you will discover that `Integer i1 = 127;` is converted to `Integer i1 = Integer.valueOf(127);` and `Integer i2 = 127;` is converted to `Integer i2 = Integer.valueOf(127);`. According to `valueOf()`'s Java documentation, this method takes advantage of caching to improve performance.

**Note** valueOf() is also used when adding a primitive-type value to a collection. For example, `col.add(27)` is converted to `col.add(Integer.valueOf(27))`.

Integer maintains an internal cache of unique Integer objects over a small range of values. The low bound of this range is -128, and the high bound defaults to 127. However, you can change the high bound by assigning a different value to system property `java.lang.Integer.IntegerCache.high` (via the `java.lang.System` class's `setProperty(String name, String value)` method—I demonstrated this method's `getProperty(String name)` counterpart in Chapter 7).
Note  Each of `java.lang.Byte`, `java.lang.Long`, and `java.lang.Short` also maintains an internal cache of unique Byte, Long, and Short objects, respectively.

Because of the cache, each `Integer.valueOf(127)` call returns the same `Integer` object reference, which is why `i1 == i2` (which compares references) evaluates to true. Because 30000 lies outside of the default range, each `Integer.valueOf(30000)` call returns a reference to a new Integer object, which is why `i1 == i2` evaluates to false.

In contrast to `==` and `!=`, which don’t unbox the boxed values prior to the comparison, operators such as `<`, `>`, and `+` unbox these values before performing their operations. As a result, `i1 < i2` is converted to `i1.intValue() < i2.intValue()` and `i1 + i2` is converted to `i1.intValue() + i2.intValue()`.

Caution  Don’t assume that autoboxing and unboxing are used in the context of the `==` and `!=` operators.

Exploring Lists

A list is an ordered collection, which is also known as a sequence. Elements can be stored in and accessed from specific locations via integer indexes. Some of these elements may be duplicates or null (when the list's implementation allows null elements). Lists are described by the `List` interface, whose generic type is `List<E>`.

List extends `Collection` and redeclares its inherited methods, partly for convenience. It also redeclares `iterator()`, `add()`, `remove()`, `equals()`, and `hashCode()` to place extra conditions on their contracts. For example, List’s contract for `add()` specifies that it appends an element to the end of the list rather than adding the element to the collection.

List also declares Table 9-2’s list-specific methods.
### Table 9-2. List-Specific Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(int index, E e)</td>
<td>Inserts element e into this list at position index. Shifts the element currently at this position (if any) and any subsequent elements to the right. This method throws UnsupportedOperationException when this list doesn’t support add(), ClassCastException when e’s class is inappropriate for this list, IllegalArgumentException when some property of e prevents it from being added to this list, NullPointerException when e contains the null reference and this list doesn’t support null elements, and java.lang.IndexOutOfBoundsException when index is less than 0 or index is greater than size().</td>
</tr>
<tr>
<td>boolean addAll(int index, Collection&lt;? extends E&gt; c)</td>
<td>Inserts all of c’s elements into this list starting at position index and in the order that they are returned by c’s iterator. Shifts the element currently at this position (if any) and any subsequent elements to the right. This method throws UnsupportedOperationException when this list doesn’t support addAll(), ClassCastException when the class of one of c’s elements is inappropriate for this list, IllegalArgumentException when some property of an element prevents it from being added to this list, NullPointerException when c contains the null reference or when one of its elements is null and this list doesn’t support null elements, and IndexOutOfBoundsException when index is less than 0 or index is greater than size().</td>
</tr>
<tr>
<td>E get(int index)</td>
<td>Returns the element stored in this list at position index. This method throws IndexOutOfBoundsException when index is less than 0 or index is greater than or equal to size().</td>
</tr>
<tr>
<td>int indexOf(Object o)</td>
<td>Returns the index of the first occurrence of element o in this list, or -1 when this list doesn’t contain the element. This method throws ClassCastException when o’s class is inappropriate for this list and NullPointerException when o contains the null reference and this list doesn’t support null elements.</td>
</tr>
<tr>
<td>int lastIndexOf(Object o)</td>
<td>Returns the index of the last occurrence of element o in this list, or -1 when this list doesn’t contain the element. This method throws ClassCastException when o’s class is inappropriate for this list and NullPointerException when o contains the null reference and this list doesn’t support null elements.</td>
</tr>
<tr>
<td>ListIterator&lt;E&gt; listIterator()</td>
<td>Returns a list iterator over the elements in this list. The elements are returned in the same order as they appear in the list.</td>
</tr>
<tr>
<td>ListIterator&lt;E&gt; listIterator(int index)</td>
<td>Returns a list iterator over the elements in this list starting with the element located at index. The elements are returned in the same order as they appear in the list. This method throws IndexOutOfBoundsException when index is less than 0 or index is greater than size().</td>
</tr>
</tbody>
</table>

(continued)
Table 9-2. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E remove(int index)</td>
<td>Removes the element at position index from this list, shifts any subsequent elements to the left, and returns this element. This method throws UnsupportedOperationException when this list doesn’t support remove() and IndexOutOfBoundsException when index is less than 0 or index is greater than or equal to size().</td>
</tr>
<tr>
<td>E set(int index, E e)</td>
<td>Replaces the element at position index in this list with element e and returns the element previously stored at this position. This method throws UnsupportedOperationException when this list doesn’t support set(), ClassCastException when e’s class is inappropriate for this list, IllegalArgumentException when some property of e prevents it from being added to this list, NullPointerException when e contains the null reference and this list doesn’t support null elements, and IndexOutOfBoundsException when index is less than 0 or index is greater than or equal to size().</td>
</tr>
<tr>
<td>List&lt;E&gt; subList(int fromIndex, int toIndex)</td>
<td>Returns a view (discussed later) of the portion of this list between fromIndex (inclusive) and toIndex (exclusive). (If fromIndex and toIndex are equal, the returned list is empty.) The returned list is backed by this list, so nonstructural changes in the returned list are reflected in this list and vice versa. The returned list supports all of the optional list methods (those methods that can throw UnsupportedOperationException) supported by this list. This method throws IndexOutOfBoundsException when fromIndex is less than 0, toIndex is greater than size(), or fromIndex is greater than toIndex.</td>
</tr>
</tbody>
</table>

Table 9-2 refers to the ListIterator interface, which is more flexible than its Iterator superinterface in that ListIterator provides methods for iterating over a list in either direction, modifying the list during iteration, and obtaining the iterator’s current position in the list.

**Note** The Iterator and ListIterator instances that are returned by the iterator() and listIterator() methods in the ArrayList and LinkedList List implementation classes are **fail-fast**: when a list is structurally modified (by calling the implementation’s add() method to add a new element, for example) after the iterator is created, in any way except through the iterator’s own add() and remove() methods, the iterator throws ConcurrentModificationException. Therefore, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, nondeterministic behavior at an undetermined time in the future.
ListIterator declares the following methods:

- **void add(E e)** inserts e into the list being iterated over. This element is inserted immediately before the next element that would be returned by next(), if any, and after the next element that would be returned by previous(), if any. This method throws UnsupportedOperationException when this list iterator doesn’t support add(), ClassCastException when e’s class is inappropriate for the list, and IllegalArgumentException when some property of e prevents it from being added to the list.

- **boolean hasNext()** returns true when this list iterator has more elements when traversing the list in the forward direction.

- **boolean hasPrevious()** returns true when this list iterator has more elements when traversing the list in the reverse direction.

- **E next()** returns the next element in the list and advances the cursor position. This method throws NoSuchElementException when there is no next element.

- **int nextIndex()** returns the index of the element that would be returned by a subsequent call to next(), or the size of the list when at the end of the list.

- **E previous()** returns the previous element in the list and moves the cursor position backward. This method throws NoSuchElementException when there is no previous element.

- **int previousIndex()** returns the index of the element that would be returned by a subsequent call to previous() or -1 when at the beginning of the list.

- **void remove()** removes from the list the last element that was returned by next() or previous(). This call can be made only once per call to next() or previous(). Furthermore, it can be made only when add() has not been called after the last call to next() or previous(). This method throws UnsupportedOperationException when this list iterator doesn’t support remove() and IllegalStateException when neither next() nor previous() has been called or remove() or add() has already been called after the last call to next() or previous().

- **void set(E e)** replaces the last element returned by next() or previous() with element e. This call can be made only when neither remove() nor add() has been called after the last call to next() or previous(). This method throws UnsupportedOperationException when this list iterator doesn’t support set(), ClassCastException when e’s class is inappropriate for the list, IllegalArgumentException when some property of e prevents it from being added to the list, and IllegalStateException when neither next() nor previous() has been called or remove() or add() has already been called after the last call to next() or previous().

A ListIterator instance doesn’t have the concept of a current element. Instead, it has the concept of a cursor for navigating through a list. The nextIndex() and previousIndex() methods return the cursor position, which always lies between the element that would be returned by a call to
previous() and the element that would be returned by a call to next(). A list iterator for a list of length n has n+1 possible cursor positions as illustrated by each caret (^) in the following:

```
Element(0)   Element(1)   Element(2)   ... Element(n-1)
cursor positions:  ^            ^            ^            ^                  ^
```

**Note** You can mix calls to next() and previous() as long as you are careful. Keep in mind that the first call to previous() returns the same element as the last call to next(). Furthermore, the first call to next() following a sequence of calls to previous() returns the same element as the last call to previous().

Table 9-2’s description of the subList() method refers to the concept of a view, which is a list that is backed by another list. Changes that are made to the view are reflected in this backing list. The view can cover the entire list or, as subList()’s name implies, only part of the list.

The subList() method is useful for performing range-view operations over a list in a compact manner. For example, list.subList(fromIndex, toIndex).clear(); removes a range of elements from list where the first element is at fromIndex and the last element is at toIndex - 1.

**Caution** A view’s meaning becomes undefined when changes are made to the backing list. Therefore, you should only use subList() temporarily whenever you need to perform a sequence of range operations on the backing list.

**ArrayList**

The ArrayList class provides a list implementation that is based on an internal array. As a result, access to the list’s elements is fast. However, because elements must be moved to open a space for insertion or to close a space after deletion, insertions and deletions of elements is slow.

ArrayList supplies three constructors:

- ArrayList() creates an empty array list with an initial capacity (storage space) of 10 elements. Once this capacity is reached, a larger array is allocated, elements from the current array are copied into the larger array, and the larger array becomes the new current array. This process repeats as additional elements are added to the array list.

- ArrayList(Collection<? extends E> c) creates an array list containing c’s elements in the order in which they are returned by c’s iterator. NullPointerException is thrown when c contains the null reference.

- ArrayList(int initialCapacity) creates an empty array list with an initial capacity of initialCapacity elements. IllegalArgumentException is thrown when initialCapacity is negative.
Listing 9-1 demonstrates an array list.

Listing 9-1. A Demonstration of an Array-Based List

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo
{
    public static void main(String[] args)
    {
        List<String> ls = new ArrayList<String>();
        String[] weekDays = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
        for (String weekDay: weekDays)
            ls.add(weekDay);
        dump("ls:", ls);
        ls.set(ls.indexOf("Wed"), "Wednesday");
        dump("ls:", ls);
        ls.remove(ls.lastIndexOf("Fri"));
        dump("ls:", ls);
    }

    static void dump(String title, List<String> ls)
    {
        System.out.print(title + " ");
        for (String s: ls)
            System.out.print(s + " ");
        System.out.println();
    }
}
```

ArrayListDemo creates an array list and an array of short weekday names. It then populates this list with these names, dumps the list to standard output, changes one of the list entries, dumps the list again, removes a list entry, and dumps the list one last time. The `dump()` method's enhanced for loop statement uses `iterator()`, `hasNext()`, and `next()` behind the scenes.

When you run this application, it generates the following output:

```
ls: Sun Mon Tue Wed Thu Fri Sat
ls: Sun Mon Tue Wednesday Thu Fri Sat
ls: Sun Mon Tuesday Thu Fri Sat
```

LinkedList

The LinkedList class provides a list implementation that is based on linked nodes. Because links must be traversed, access to the list's elements is slow. However, because only node references need to be changed, insertions and deletions of elements are fast.
WHAT IS A NODE?

A *node* is a fixed sequence of value and link memory locations. Unlike an array, where each slot stores a single value of the same primitive type or reference supertype, a node can store multiple values of different types. It can also store *links* (references to other nodes).

Consider the following simple Node class:

```java
class Node {
    String name; // value field
    Node next;   // link field
}
```

Node describes simple nodes where each node consists of a single name value field and a single next link field. Notice that next is of the same type as the class in which it is declared. This arrangement lets a node instance store a reference to another node instance (which is the next node) in this field. The resulting nodes are linked together.

The following code fragment creates two Node objects and links the second Node object to the first Node object. This fragment also demonstrates how to traverse this linked list by following each Node object’s next field. Node traversal stops when the traversal code discovers that next contains the null reference, which signifies the end of the list:

```java
Node first = new Node();
first.name = "First node"; // You would normally provide getter and setter methods.
Node last = new Node();
last.name = "Last node";
last.next = null;
first.next = last;
Node temp = first;
while (temp != null) {
    System.out.println(temp.name);
    temp = temp.next;
}
```

The code first builds a linked list of two Nodes and then assigns first to local variable temp in order to traverse the list without losing the reference to the first node that is stored in first. While temp is not null, the loop outputs the name field. It also navigates to the next Node object in the list via the temp = temp.next; statement.

If you convert this code into an application and run the application, you will discover the following output:

```
First node
Last node
```
LinkedList supplies two constructors:

- LinkedList() creates an empty linked list.
- LinkedList(Collection<? extends E> c) creates a linked list containing c's elements in the order in which they are returned by c's iterator. NullPointerException is thrown when c contains the null reference.

Listing 9-2 demonstrates a linked list.

Listing 9-2. A Demonstration of a List of Linked Nodes

```java
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class LinkedListDemo
{
    public static void main(String[] args)
    {
        List<String> ls = new LinkedList<String>();
        String[] weekDays = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
        for (String weekDay: weekDays)
            ls.add(weekDay);
        LinkedListDemo.dump("ls:", ls);
        ls.add(1, "Sunday");
        ls.add(3, "Monday");
        ls.add(5, "Tuesday");
        ls.add(7, "Wednesday");
        ls.add(9, "Thursday");
        ls.add(11, "Friday");
        ls.add(13, "Saturday");
        LinkedListDemo.dump("ls:", ls);
        ListIterator<String> li = ls.listIterator(ls.size());
        while (li.hasPrevious())
            System.out.print(li.previous() + " ");
        System.out.println();
    }

    static void dump(String title, List<String> ls)
    {
        System.out.print(title + " ");
        for (String s: ls)
            System.out.print(s + " ");
        System.out.println();
    }
}
```

LinkedListDemo creates a linked list and an array of short weekday names. It then populates this list with these names, dumps the list to standard output, inserts longer weekday names after their shorter name counterparts, dumps the list again, and outputs the list in reverse order by using a list iterator with its cursor initialized past the list's end and repeatedly calling its previous() method.
When you run this application, it generates the following output:

```
ls: Sun Mon Tue Wed Thu Fri Sat
ls: Sun Sunday Mon Monday Tue Tuesday Wed Wednesday Thu Thursday Fri Friday Sat Saturday
Saturday Sat Friday Fri Thursday Thu Wednesday Wed Tuesday Tue Monday Mon Sunday Sun
```

## Exploring Sets

A set is a collection that contains no duplicate elements. In other words, a set contains no pair of elements $e_1$ and $e_2$ such that $e_1.equals(e_2)$ returns true. Furthermore, a set can contain at most one null element. Sets are described by the `Set` interface, whose generic type is `Set<E>`.

Set extends `Collection` and redeclares its inherited methods, for convenience and also to add stipulations to the contracts for `add()`, `equals()`, and `hashCode()` to address how they behave in a set context. Also, Set’s documentation states that all constructors of implementation classes must create sets that contain no duplicate elements.

Set doesn’t introduce new methods.

### TreeSet

The `TreeSet` class provides a set implementation that is based on a tree data structure. As a result, elements are stored in sorted order. However, accessing these elements is somewhat slower than with the other `Set` implementations (which are not sorted) because links must be traversed.


TreeSet supplies four constructors:

- `TreeSet()` creates a new, empty tree set that is sorted according to the natural ordering of its elements. All elements inserted into the set must implement the `Comparable` interface.
- `TreeSet(Collection<? extends E> c)` creates a new tree set containing c’s elements, sorted according to the natural ordering of its elements. All elements inserted into the new set must implement the `Comparable` interface. This constructor throws `ClassCastException` when c’s elements don’t implement `Comparable` or are not mutually comparable and `NullPointerException` when c contains the null reference.
- `TreeSet(Comparator<? super E> comparator)` creates a new, empty tree set that is sorted according to the specified `comparator`. Passing null to comparator implies that natural ordering will be used.
- `TreeSet(SortedSet<E> ss)` creates a new tree set containing the same elements and using the same ordering as ss. (I discuss sorted sets later in this chapter.) This constructor throws `NullPointerException` when ss contains the null reference.
Listing 9-3 demonstrates a tree set.

Listing 9-3. A Demonstration of a Tree Set with String Elements Sorted According to Their Natural Ordering

```java
import java.util.Set;
import java.util.TreeSet;

public class TreeSetDemo
{
    public static void main(String[] args)
    {
        Set<String> ss = new TreeSet<String>();
        String[] fruits = {"apples", "pears", "grapes", "bananas", "kiwis"};
        for (String fruit: fruits)
            ss.add(fruit);
        dump("ss:", ss);
    }

    static void dump(String title, Set<String> ss)
    {
        System.out.print(title + " ");
        for (String s: ss)
            System.out.print(s + " ");
        System.out.println();
    }
}
```

TreeSetDemo creates a tree set and an array of fruit names. It then populates this set with these names and dumps the set to standard output. Because String implements Comparable, it's legal for this application to insert the contents of the fruits array into a tree set that was created via the TreeSet() constructor.

When you run this application, it generates the following output:

ss: apples bananas grapes kiwis pears

**HashSet**

The HashSet class provides a set implementation that is backed by a hashtable data structure (implemented as a HashMap instance, discussed later, which provides a quick way to determine if an element has already been stored in this structure). Although this class provides no ordering guarantees for its elements, HashSet is much faster than TreeSet. Furthermore, HashSet permits the null reference to be stored in its instances.

**Note** Check out Wikipedia’s “Hash table” entry (http://en.wikipedia.org/wiki/Hash_table) to learn about hashtables.
HashSet supplies four constructors:

- `HashSet()` creates a new, empty hashset where the backing HashMap instance has an initial capacity of 16 and a load factor of 0.75. You will learn what these items mean when I discuss HashMap later in this chapter.

- `HashSet(Collection<? extends E> c)` creates a new hashset containing `c`'s elements. The backing HashMap has an initial capacity sufficient to contain `c`'s elements and a load factor of 0.75. This constructor throws `NullPointerException` when `c` contains the null reference.

- `HashSet(int initialCapacity)` creates a new, empty hashset where the backing HashMap instance has the capacity specified by `initialCapacity` and a load factor of 0.75. This constructor throws `IllegalArgumentException` when `initialCapacity`'s value is less than 0.

- `HashSet(int initialCapacity, float loadFactor)` creates a new, empty hashset where the backing HashMap instance has the capacity specified by `initialCapacity` and the load factor specified by `loadFactor`. This constructor throws `IllegalArgumentException` when `initialCapacity` is less than 0 or when `loadFactor` is less than or equal to 0.

Listing 9-4 demonstrates a hashset.

**Listing 9-4. A Demonstration of a Hashset with String Elements Unordered**

```java
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo
{
    public static void main(String[] args)
    {
        Set<String> ss = new HashSet<String>();
        String[] fruits = {"apples", "pears", "grapes", "bananas", "kiwis", "pears", null};
        for (String fruit: fruits)
            ss.add(fruit);
        dump("ss:", ss);
    }

    static void dump(String title, Set<String> ss)
    {
        System.out.print(title + " ");
        for (String s: ss)
            System.out.print(s + " ");
        System.out.println();
    }
}
```

HashSetDemo creates a hashset and an array of fruit names. It then populates this set with these names and dumps the set to standard output. Unlike with TreeSet, HashSet permits null to be added (NullPointerException isn't thrown), which is why Listing 9-4 includes null in HashSetDemo's fruits array.
When you run this application, it generates unordered output such as the following:

ss: null grapes bananas kiwis pears apples

Suppose you want to add instances of your classes to a hashset. As with String, your classes must override equals() and hashCode(); otherwise, duplicate class instances can be stored in the hashset. For example, Listing 9-5 presents the source code to an application whose Planet class overrides equals() but fails to also override hashCode().

Listing 9-5. A Custom Planet Class Not Overriding hashCode()

```java
import java.util.HashSet;
import java.util.Set;

public class CustomClassAndHashSet
{
    public static void main(String[] args)
    {
        Set<Planet> sp = new HashSet<Planet>();
        sp.add(new Planet("Mercury"));
        sp.add(new Planet("Venus"));
        sp.add(new Planet("Earth"));
        sp.add(new Planet("Mars"));
        sp.add(new Planet("Jupiter"));
        sp.add(new Planet("Saturn"));
        sp.add(new Planet("Uranus"));
        sp.add(new Planet("Neptune"));
        sp.add(new Planet("Fomalhaut b"));
        Planet p1 = new Planet("51 Pegasi b");
        sp.add(p1);
        Planet p2 = new Planet("51 Pegasi b");
        sp.add(p2);
        System.out.println(p1.equals(p2));
        System.out.println(sp);
    }
}

class Planet
{
    private String name;

    Planet(String name)
    {
        this.name = name;
    }

    @Override
    public boolean equals(Object o)
    {
        if (!(o instanceof Planet))
            return false;
        return this.name.equals(((Planet) o).name);
    }

    @Override
    public int hashCode()
    {
        return name.hashCode();
    }
}
```

www.it-ebooks.info
Listing 9-5’s Planet class declares a single name field of type String. Although it might seem pointless to declare Planet with a single String field because I could refactor this listing to remove Planet and work with String, I might want to introduce additional fields to Planet (perhaps to store a planet’s mass and other characteristics) in the future.

When you run this application, it generates unordered output such as the following:

```
true
[Neptune, Mars, Mercury, Fomalhaut b, Venus, 51 Pegasi b, 51 Pegasi b, Jupiter, Saturn, Earth, Uranus]
```

This output reveals two 51 Pegasi b elements in the hashset. Although these elements are equal from the perspective of the overriding equals() method (the first output line, true, proves this point), overriding equals() isn’t enough to avoid duplicate elements being stored in a hashset: you must also override hashCode().

The easiest way to override hashCode() in Listing 9-5’s Planet class is to have the overriding method call the name field’s hashCode() method and return its value. (This technique only works with a class whose single reference field’s class provides a valid hashCode() method.) Listing 9-6 presents this overriding hashCode() method.

Listing 9-6. A Custom Planet Class Overriding hashCode()

```java
import java.util.HashSet;
import java.util.Set;

public class CustomClassAndHashSet
{
    public static void main(String[] args)
    {
        Set<Planet> sp = new HashSet<Planet>();
        sp.add(new Planet("Mercury"));
        sp.add(new Planet("Venus"));
        sp.add(new Planet("Earth"));
        sp.add(new Planet("Mars"));
        sp.add(new Planet("Jupiter"));
        sp.add(new Planet("Saturn"));
```
sp.add(new Planet("Uranus"));
sp.add(new Planet("Neptune"));
sp.add(new Planet("Fomalhaut b"));
Planet p1 = new Planet("51 Pegasi b");
sp.add(p1);
Planet p2 = new Planet("51 Pegasi b");
sp.add(p2);
System.out.println(p1.equals(p2));
System.out.println(sp);
}
}

class Planet
{
    private String name;

    Planet(String name)
    {
        this.name = name;
    }

    @Override
    public boolean equals(Object o)
    {
        if (!(o instanceof Planet))
            return false;
        Planet p = (Planet) o;
        return p.name.equals(name);
    }

    String getName()
    {
        return name;
    }

    @Override
    public int hashCode()
    {
        return name.hashCode();
    }

    @Override
    public String toString()
    {
        return name;
    }
}
Compile Listing 9-6 (javac CustomClassAndHashSet.java) and run the application (java CustomClassAndHashSet). You will observe output (similar to that shown below) that reveals no duplicate elements:

true

[Saturn, Earth, Uranus, Fomalhaut b, 51 Pegasi b, Venus, Jupiter, Mercury, Mars, Neptune]

**Note** LinkedHashSet is a subclass of HashSet that uses a linked list to store its elements. As a result, LinkedHashSet’s iterator returns elements in the order in which they were inserted. For example, if Listing 9-4 had specified Set<String> ss = new LinkedHashSet<String>();, the application’s output would have been ss: apples pears grapes bananas kiwis null. Also, LinkedHashSet offers slower performance than HashSet and faster performance than TreeSet.

**EnumSet**

In Chapter 6, I introduced you to traditional enumerated types and their enum replacement. (An enum is an enumerated type that is expressed via reserved word enum.) The following example demonstrates the traditional enumerated type:

```java
static final int SUNDAY = 1;
static final int MONDAY = 2;
static final int TUESDAY = 4;
static final int WEDNESDAY = 8;
static final int THURSDAY = 16;
static final int FRIDAY = 32;
static final int SATURDAY = 64;
```

Although the enum has many advantages over the traditional enumerated type, the traditional enumerated type is less awkward to use when combining constants into a set, for example,

```
static final int DAYS_OFF = SUNDAY | MONDAY;
```

DAYS_OFF is an example of an integer-based, fixed-length *bitset*, which is a set of bits where each bit indicates that its associated member belongs to the set when the bit is set to 1 and is absent from the set when the bit is set to 0.

**Note** An int-based bitset cannot contain more than 32 members because int has a size of 32 bits. Similarly, a long-based bitset cannot contain more than 64 members because long has a size of 64 bits.

This bitset is formed by bitwise inclusive ORing the traditional enumerated type’s integer constants together via the bitwise inclusive OR operator (|): you could also use +. Each constant must be a unique power of two (starting with one) because otherwise it’s impossible to distinguish between the members of this bitset.
To determine if a constant belongs to the bitset, create an expression that involves the bitwise AND operator (&). For example, `((DAYS_OFF & MONDAY) == MONDAY)` bitwise ANDs DAYS_OFF (3) with MONDAY (2), which results in 2. This value is compared via `==` with MONDAY (2), and the result of the expression is true: MONDAY is a member of the DAYS_OFF bitset.

You can accomplish the same task with an enum by instantiating an appropriate Set implementation class and calling the `add()` method multiple times to store the constants in the set. Listing 9-7 illustrates this more awkward alternative.

**Listing 9-7. Creating the Set Equivalent of DAYS_OFF**

```java
import java.util.Set;
import java.util.TreeSet;

enum Weekday
{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class DaysOff
{
    public static void main(String[] args)
    {
        Set<Weekday> daysOff = new TreeSet<Weekday>();
        daysOff.add(Weekday.SUNDAY);
        daysOff.add(Weekday.MONDAY);
        System.out.println(daysOff);
    }
}
```

When you run this application, it generates the following output:

```
[SUNDAY, MONDAY]
```

**Note** The constants’ ordinals and not their names are stored in the tree set, which is why the names appear unordered (S before M) even though the constants are stored in sorted order of their ordinals.

As well as being more awkward to use (and verbose) than the bitset, the Set alternative requires more memory to store each constant and isn’t as fast. Because of these problems, EnumSet was introduced.

The EnumSet class provides a Set implementation that is based on a bitset. Its elements are constants that must come from the same enum, which is specified when the enum set is created. Null elements are not permitted; any attempt to store a null element results in a thrown NullPointerException.
Listing 9-8 demonstrates EnumSet.

Listing 9-8. Creating the EnumSet Equivalent of DAYS_OFF

```java
import java.util.EnumSet;
import java.util.Iterator;
import java.util.Set;

enum Weekday {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumSetDemo {
    public static void main(String[] args) {
        Set<Weekday> daysOff = EnumSet.of(Weekday.SUNDAY, Weekday.MONDAY);
        Iterator<Weekday> iter = daysOff.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

EnumSetDemo takes advantage of the fact that EnumSet, whose generic type is EnumSet<E extends Enum<E>>, provides various class methods for conveniently constructing enum sets. For example, `EnumSet<E extends Enum<E>> of(E e1, E e2)` returns an EnumSet instance consisting of elements e1 and e2. In this example, those elements are Weekday.SUNDAY and Weekday.MONDAY.

When you run this application, it generates the following output:

SUNDAY
MONDAY

Note As well as providing several overloaded `of()` methods, EnumSet provides other methods for conveniently creating enum sets. For example, `allOf()` returns an EnumSet instance that contains all of an enum’s constants, where this method’s solitary argument is a class literal (an expression consisting of a class’s name followed by a dot followed by reserved word `class`) that identifies the enum:

```java
Set<Weekday> allWeekDays = EnumSet.allOf(Weekday.class);
```

Similarly, `range()` returns an EnumSet instance containing a range of an enum’s elements (with the range’s limits as specified by this method’s two arguments):

```java
for (WeekDay wd: EnumSet.range(WeekDay.MONDAY, WeekDay.FRIDAY))
    System.out.println(wd);
```
Exploring Sorted Sets

TreeSet is an example of a sorted set, which is a set that maintains its elements in ascending order, sorted according to their natural ordering or according to a comparator that is supplied when the sorted set is created. Sorted sets are described by the SortedSet interface.

SortedSet, whose generic type is SortedSet<E>, extends Set. With two exceptions, the methods it inherits from Set behave identically on sorted sets as on other sets:

- The Iterator instance returned from iterator() traverses the sorted set in ascending element order.
- The array returned by toArray() contains the sorted set's elements in order.

Note Although not guaranteed, the toString() methods of SortedSet implementations in the Collections Framework (such as TreeSet) return a string containing all of the sorted set's elements in order.

SortedSet’s documentation requires that an implementation provide the four standard constructors that I presented in my discussion of TreeSet. Furthermore, implementations of this interface must implement the methods that are described in Table 9-3.

Table 9-3. SortedSet-Specific Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparator&lt;? super E&gt; comparator()</td>
<td>Returns the comparator used to order the elements in this set or null when this set uses the natural ordering of its elements.</td>
</tr>
<tr>
<td>E first()</td>
<td>Returns the first (lowest) element currently in this set, or throws a NoSuchElementException instance when this set is empty.</td>
</tr>
<tr>
<td>SortedSet&lt;E&gt; headSet(E toElement)</td>
<td>Returns a view of that portion of this set whose elements are strictly less than toElement. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws ClassCastException when toElement is not compatible with this set’s comparator (or, when the set has no comparator, when toElement doesn't implement Comparable), NullPointerException when toElement is null and this set doesn't permit null elements, and IllegalArgumentException when this set has a restricted range and toElement lies outside of this range's bounds.</td>
</tr>
<tr>
<td>E last()</td>
<td>Returns the last (highest) element currently in this set, or throws a NoSuchElementException instance when this set is empty.</td>
</tr>
</tbody>
</table>

(continued)
Note Unlike a list-based range view whose endpoints are elements in the backing list, the endpoints of a set-based range view are absolute points in element space, allowing a set-based range view to serve as a window onto a portion of the set’s element space. Any changes made to the set-based range view are written back to the backing sorted set and vice versa.

Each range view returned by headSet(), subSet(), or tailSet() is half open because it doesn’t include its high endpoint (headSet() and subSet()) or its low endpoint (tailSet()). For the first two methods, the high endpoint is specified by argument toElement; for the last method, the low endpoint is specified by argument fromElement.

Note You could also regard the returned range view as being half closed because it includes only one of its endpoints.
Listing 9-9 demonstrates a sorted set based on a tree set.

Listing 9-9. A Sorted Set of Fruit and Vegetable Names

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo
{
    public static void main(String[] args)
    {
        SortedSet<String> sss = new TreeSet<String>();
        String[] fruitAndVeg =
        {
            "apple", "potato", "turnip", "banana", "corn", "carrot", "cherry",
            "pear", "mango", "strawberry", "cucumber", "grape", "banana",
            "kiwi", "radish", "blueberry", "tomato", "onion", "raspberry",
            "lemon", "pepper", "squash", "melon", "zucchini", "peach", "plum",
            "turnip", "onion", "nectarine"
        };
        System.out.println("Array size = " + fruitAndVeg.length);
        for (String fruitVeg: fruitAndVeg)
            sss.add(fruitVeg);
        dump("sss: ", sss);
        System.out.println("Sorted set size = " + sss.size());
        System.out.println("First element = " + sss.first());
        System.out.println("Last element = " + sss.last());
        System.out.println("Comparator = " + sss.comparator());
        dump("hs: ", sss.headSet("n"));
        dump("ts: ", sss.tailSet("n"));
        System.out.println("Count of p-named fruits & vegetables = " +
                sss.subSet("p", "q").size());
        System.out.println("Incorrect count of c-named fruits & vegetables = " +
                sss.subSet("carrot", "cucumber").size());
        System.out.println("Correct count of c-named fruits & vegetables = " +
                sss.subSet("carrot", "cucumber\0").size());
    }
    static void dump(String title, SortedSet<String> sss)
    {
        System.out.print(title + " ");
        for (String s: sss)
            System.out.print(s + " ");
        System.out.println();
    }
}
```

SortedSetDemo creates a sorted set and an array of fruit and vegetable names and then proceeds to populate the set from this array. After dumping out the set's contents, it outputs information about the set, including head and tail views of portions of the set.
When you run this application, it generates the following output:

Array size = 29
sss: apple banana blueberry carrot cherry corn cucumber grape kiwi lemon mango melon nectarine onion peach pear pepper plum potato radish raspberry squash strawberry tomato turnip zucchini
Sorted set size = 26
First element = apple
Last element = zucchini
Comparator = null
hs: apple banana blueberry carrot cherry corn cucumber grape kiwi lemon mango melon
ts: nectarine onion peach pear pepper plum potato radish raspberry squash strawberry tomato turnip zucchini
Count of p-named fruits & vegetables = 5
Incorrect count of c-named fruits & vegetables = 3
Correct count of c-named fruits & vegetables = 4

This output reveals that the sorted set’s size is less than the array’s size because a set cannot contain duplicate elements: the duplicate banana, turnip, and onion elements are not stored in the sorted set.

The comparator() method returns null because the sorted set was not created with a comparator. Instead, the sorted set relies on the natural ordering of String elements to store them in sorted order.

The headSet() and tailSet() methods are called with argument "n" to return, respectively, a set of elements whose names begin with a letter that is strictly less than n, and a letter that is greater than or equal to n.

Finally, the output shows you that you must be careful when passing an upper limit to subSet(). As you can see, ss.subSet("carrot", "cucumber") doesn’t include cucumber in the returned range view because cucumber is subSet()’s high endpoint.

To include cucumber in the range view, you need to form a closed range or closed interval (both endpoints are included). With String objects, you accomplish this task by appending \0 to the string. For example, ss.subSet("carrot", "cucumber\0") includes cucumber because it is less than cucumber\0.

This same technique can be applied wherever you need to form an open range or open interval (neither endpoint is included). For example, ss.subSet("carrot\0", "cucumber") doesn’t include carrot because it is less than carrot\0. Furthermore, it doesn’t include high endpoint cucumber.

Note  When you want to create closed and open ranges for elements created from your own classes, you need to provide some form of predecessor() and successor() methods that return an element’s predecessor and successor.

You need to be careful when designing classes that work with sorted sets. For example, the class must implement Comparable when you plan to store the class’s instances in a sorted set where these elements are sorted according to their natural ordering. Consider Listing 9-10.
Listing 9-10. A Custom Employee Class Not Implementing Comparable

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class CustomClassAndSortedSet
{
    public static void main(String[] args)
    {
        SortedSet<Employee> sse = new TreeSet<Employee>();
        sse.add(new Employee("Sally Doe"));
        sse.add(new Employee("Bob Doe")); // ClassCastException thrown here
        sse.add(new Employee("John Doe"));
        System.out.println(sse);
    }
}

class Employee
{
    private String name;

    Employee(String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

When you run this application, it generates the following output:

```
Exception in thread "main" java.lang.ClassCastException: Employee cannot be cast to java.lang.Comparable
    at java.util.TreeMap.compare(Unknown Source)
    at java.util.TreeMap.put(Unknown Source)
    at java.util.TreeSet.add(Unknown Source)
    at CustomClassAndSortedSet.main(CustomClassAndSortedSet.java:9)
```

The ClassCastException instance is thrown during the second `add()` method call because the sorted set implementation, an instance of TreeSet, is unable to call the second Employee element's `compareTo()` method, because Employee doesn't implement Comparable.

The solution to this problem is to have the class implement Comparable, which is exactly what is revealed in Listing 9-11.
Listing 9-11. Making Employee Elements Comparable

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class CustomClassAndSortedSet
{
    public static void main(String[] args)
    {
        SortedSet<Employee> sse = new TreeSet<Employee>();
        sse.add(new Employee("Sally Doe"));
        sse.add(new Employee("Bob Doe"));
        Employee e1 = new Employee("John Doe");
        Employee e2 = new Employee("John Doe");
        sse.add(e1);
        sse.add(e2);
        System.out.println(sse);
        System.out.println(e1.equals(e2));
    }
}

class Employee implements Comparable<Employee>
{
    private String name;

    Employee(String name)
    {
        this.name = name;
    }

    @Override
    public int compareTo(Employee e)
    {
        return name.compareTo(e.name);
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

Listing 9-11’s `main()` method differs from Listing 9-10 in that it also creates two `Employee` objects initialized to "John Doe," adds these objects to the sorted set, and compares these objects for equality via `equals()`. Furthermore, Listing 9-11 declares `Employee` to implement `Comparable`, introducing a `compareTo()` method into `Employee`.

When you run this application, it generates the following output:

```
[Bob Doe, John Doe, Sally Doe]
false
```
This output shows that only one "John Doe" Employee object is stored in the sorted set. After all, a set cannot contain duplicate elements. However, the false value (resulting from the equals() comparison) also shows that the sorted set's natural ordering is inconsistent with equals(), which violates SortedSet's contract:

The ordering maintained by a sorted set (whether or not an explicit comparator is provided) must be consistent with equals() if the sorted set is to correctly implement the Set interface. This is so because the Set interface is defined in terms of the equals() operation, but a sorted set performs all element comparisons using its compareTo() (or compare()) method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal.

Because the application works correctly, why should SortedSet's contract matter? Although the contract doesn't appear to matter with respect to the TreeSet implementation of SortedSet, perhaps it will matter in the context of a third-party class that implements this interface.

Listing 9-12 shows you how to correct this problem and make Employee instances work with any implementation of a sorted set.

Listing 9-12. A Contract-Compliant Employee Class

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class CustomClassAndSortedSet
{
    public static void main(String[] args)
    {
        SortedSet<Employee> sse = new TreeSet<Employee>();
        sse.add(new Employee("Sally Doe"));
        sse.add(new Employee("Bob Doe"));
        Employee e1 = new Employee("John Doe");
        Employee e2 = new Employee("John Doe");
        sse.add(e1);
        sse.add(e2);
        System.out.println(sse);
        System.out.println(e1.equals(e2));
    }
}

class Employee implements Comparable<Employee>
{
    private String name;
    Employee(String name)
    {
        this.name = name;
    }
    @Override
    public int compareTo(Employee e)
    {
        return name.compareTo(e.name);
    }
}
```

www.it-ebooks.info
Listing 9-12 corrects the SortedSet contract violation by overriding `equals()`. Run the resulting application and you will observe [Bob Doe, John Doe, Sally Doe] as the first line of output and true as the second line: the sorted set’s natural ordering is now consistent with `equals()`.

Note Although it’s important to override `hashCode()` whenever you override `equals()`, I didn’t override `hashCode()` (although I overrode `equals()`) in Listing 9-12’s Employee class to emphasize that tree-based sorted sets ignore `hashCode()`.

Exploring Navigable Sets

TreeSet is an example of a navigable set, which is a sorted set that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets. Navigable sets are described by the NavigableSet interface, whose generic type is `NavigableSet<E>`, which extends `SortedSet`, and which is described in Table 9-4.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E ceiling(E e)</code></td>
<td>Returns the least element in this set greater than or equal to <code>e</code>, or null when there is no such element. This method throws <code>ClassCastException</code> when <code>e</code> cannot be compared with the elements currently in the set and <code>NullPointerException</code> when <code>e</code> is null and this set doesn’t permit null elements.</td>
</tr>
<tr>
<td><code>Iterator&lt;E&gt; descendingIterator()</code></td>
<td>Returns an iterator over the elements in this set, in descending order. Equivalent in effect to <code>descendingSet().iterator()</code>.</td>
</tr>
<tr>
<td><code>NavigableSet&lt;E&gt; descendingSet()</code></td>
<td>Returns a reverse order view of the elements contained in this set. The descending set is backed by this set, so changes to the set are reflected in the descending set and vice versa. If either set is modified (except through the iterator’s own <code>remove()</code> operation) while iterating over the set, the results of the iteration are undefined.</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E floor(E e)</td>
<td>Returns the greatest element in this set less than or equal to e or null when there is no such element. This method throws ClassCastException when e cannot be compared with the elements currently in the set and NullPointerException when e is null and this set doesn't permit null elements.</td>
</tr>
<tr>
<td>NavigableSet&lt;E&gt; headSet(E toElement, boolean inclusive)</td>
<td>Returns a view of the portion of this set whose elements are less than (or equal to, when inclusive is true) toElement. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws ClassCastException when toElement is not compatible with this set's comparator (or, when the set has no comparator, when toElement doesn't implement Comparable), NullPointerException when toElement is null and this set doesn't permit null elements, and IllegalArgumentException when this set has a restricted range and toElement lies outside of this range's bounds.</td>
</tr>
<tr>
<td>E higher(E e)</td>
<td>Returns the least element in this set strictly greater than the given element or null when there is no such element. This method throws ClassCastException when e cannot be compared with the elements currently in the set and NullPointerException when e is null and this set doesn't permit null elements.</td>
</tr>
<tr>
<td>E lower(E e)</td>
<td>Returns the greatest element in this set strictly less than the given element or null when there is no such element. This method throws ClassCastException when e cannot be compared with the elements currently in the set and NullPointerException when e is null and this set doesn't permit null elements.</td>
</tr>
<tr>
<td>E pollFirst()</td>
<td>Returns and removes the first (lowest) element from this set, or returns null when this set is empty.</td>
</tr>
<tr>
<td>E pollLast()</td>
<td>Returns and removes the last (highest) element from this set, or returns null when this set is empty.</td>
</tr>
<tr>
<td>NavigableSet&lt;E&gt; subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)</td>
<td>Returns a view of the portion of this set whose elements range from fromElement to toElement. (When fromElement and toElement are equal, the returned set is empty unless fromInclusive and toInclusive are both true.) The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws ClassCastException when fromElement and toElement cannot be compared to one another using this set's comparator (or, when the set has no comparator, using natural ordering), NullPointerException when fromElement or toElement is null and this set doesn't permit null elements, and IllegalArgumentException when fromElement is greater than toElement or when this set has a restricted range and fromElement or toElement lies outside of this range's bounds.</td>
</tr>
</tbody>
</table>
Table 9-4. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NavigableSet&lt;E&gt; tailSet(E fromElement, boolean inclusive)</td>
<td>Returns a view of the portion of this set whose elements are greater than (or equal to, when inclusive is true) fromElement. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws ClassCastException when fromElement is not compatible with this set's comparator (or, when the set has no comparator, when fromElement doesn't implement Comparable), NullPointerException when fromElement is null and this set doesn't permit null elements, and IllegalArgumentException when this set has a restricted range and fromElement lies outside of this range's bounds.</td>
</tr>
</tbody>
</table>

Listing 9-13 demonstrates a navigable set based on a tree set.

Listing 9-13. Navigating a Set of Integers

```java
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.TreeSet;

public class NavigableSetDemo {
    public static void main(String[] args) {
        NavigableSet<Integer> ns = new TreeSet<Integer>();
        int[] ints = {82, -13, 4, 0, 11, -6, 9};
        for (int i : ints)
            ns.add(i);
        System.out.print("Ascending order: ");
        Iterator iter = ns.iterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");
        System.out.println();
        System.out.print("Descending order: ");
        iter = ns.descendingIterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");
        System.out.println("\n");
        outputClosestMatches(ns, 4);
        outputClosestMatches(ns.descendingSet(), 12);
    }

    static void outputClosestMatches(NavigableSet<Integer> ns, int i) {
        System.out.println("Element < " + i + " is " + ns.lower(i));
        System.out.println("Element <= " + i + " is " + ns.floor(i));
        System.out.println("Element > " + i + " is " + ns.higher(i));
        System.out.println("Element >= " + i + " is " + ns.ceiling(i));
    }
}
```
Listing 9-13 creates a navigable set of Integer elements. It takes advantage of autoboxing to ensure that ints are converted to Integers.

When you run this application, it generates the following output:

Ascending order: -13 -6 0 4 9 11 82
Descending order: 82 11 9 4 0 -6 -13

Element < 4 is 0
Element <= 4 is 4
Element > 4 is 9
Element >= 4 is 4

Element < 12 is 82
Element <= 12 is 82
Element > 12 is 11
Element >= 12 is 11

The first four output lines beginning with Element pertain to an ascending-order set where the element being matched (4) is a member of the set. The second four Element-prefixed lines pertain to a descending-order set where the element being matched (12) is not a member.

As well as letting you conveniently locate set elements via its closest-match methods (ceiling(), floor(), higher(), and lower()), NavigableSet lets you return set views containing all elements within certain ranges, as demonstrated by the following examples:

- ns.subSet(-13, true, 9, true): Returns all elements from -13 through 9.
- ns.tailSet(-6, false): Returns all elements greater than -6.
- ns.headSet(4, true): Returns all elements less than or equal to 4.

Finally, you can return and remove from the set the first (lowest) element by calling pollFirst() and the last (highest) element by calling pollLast(). For example, ns.pollFirst() removes and returns -13, and ns.pollLast() removes and returns 82.

Exploring Queues

A queue is a collection in which elements are stored and retrieved in a specific order. Most queues are categorized as one of the following:

- **First-In, First-Out (FIFO) queue**: Elements are inserted at the queue’s tail and removed at the queue’s head.
- **Last-In, First-Out (LIFO) queue**: Elements are inserted and removed at one end of the queue such that the last element inserted is the first element retrieved. This kind of queue behaves as a stack.
- **Priority queue**: Elements are inserted according to their natural ordering or according to a comparator that is supplied to the queue implementation.
CHAPTER 9: Exploring the Collections Framework

Queue, whose generic type is Queue<E>, extends Collection, redenning add() to adjust its contract (insert the specified element into this queue if it's possible to do so immediately without violating capacity restrictions), and inheriting the other methods from Collection. Table 9-5 describes add() and the other Queue-specific methods.

Table 9-5 reveals two sets of methods: in one set, a method (such as add()) throws an exception when an operation fails; in the other set, a method (such as offer()) returns a special value (false or null) in the presence of failure. The methods that return a special value are useful in the context of capacity-restricted Queue implementations where failure is a normal occurrence.

Table 9-5. Queue-Specific Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add(E e)</td>
<td>Inserts element e into this queue if it is possible to do so immediately without violating capacity restrictions. Returns true on success; otherwise, throws IllegalStateException when the element cannot be added at this time because no space is currently available. This method also throws ClassCastException when e's class prevents e from being added to this queue, NullPointerException when e contains the null reference and this queue doesn't permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this queue.</td>
</tr>
<tr>
<td>E element()</td>
<td>Returns but doesn't also remove the element at the head of this queue. This method throws NoSuchElementException when this queue is empty.</td>
</tr>
<tr>
<td>boolean offer(E e)</td>
<td>Inserts element e into this queue if it is possible to do so immediately without violating capacity restrictions. Returns true on success; otherwise, returns false when the element cannot be added at this time because no space is currently available. This method throws ClassCastException when e's class prevents e from being added to this queue, NullPointerException when e contains the null reference and this queue doesn't permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this queue.</td>
</tr>
<tr>
<td>E peek()</td>
<td>Returns but doesn't also remove the element at the head of this queue. This method returns null when this queue is empty.</td>
</tr>
<tr>
<td>E poll()</td>
<td>Returns and also removes the element at the head of this queue. This method returns null when this queue is empty.</td>
</tr>
<tr>
<td>E remove()</td>
<td>Returns and also removes the element at the head of this queue. This method throws NoSuchElementException when this queue is empty. This is the only difference between remove() and poll().</td>
</tr>
</tbody>
</table>

Note  The offer() method is generally preferable to add() when using a capacity-restricted queue because offer() doesn't throw IllegalStateException.
Java supplies many Queue implementation classes, where most of these classes are members of the java.util.concurrent package: LinkedBlockingQueue and SynchronousQueue are examples. In contrast, the java.util package provides LinkedList and PriorityQueue as its Queue implementation classes.

**Caution** Many Queue implementation classes don’t allow null elements to be added. However, some classes (such as LinkedList) permit null elements. You should avoid adding a null element because null is used as a special return value by the peek() and poll() methods to indicate that a queue is empty.

### PriorityQueue

The PriorityQueue class provides an implementation of a priority queue, which is a queue that orders its elements according to their natural ordering or by a comparator provided when the queue is instantiated. Priority queues don’t permit null elements and don’t permit insertion of non-Comparable objects when relying on natural ordering.

The element at the head of the priority queue is the least element with respect to the specified ordering. When multiple elements are tied for least element, one of those elements is arbitrarily chosen as the least element. Similarly, the element at the tail of the priority queue is the greatest element, which is arbitrarily chosen when there is a tie.

Priority queues are unbounded but have a capacity that governs the size of the internal array that is used to store the priority queue’s elements. The capacity value is at least as large as the queue’s length, and grows automatically as elements are added to the priority queue.

PriorityQueue (whose generic type is PriorityQueue<E>) supplies six constructors:

- PriorityQueue() creates a PriorityQueue instance with an initial capacity of 11 elements and which orders its elements according to their natural ordering.
- PriorityQueue(Collection<? extends E> c) creates a PriorityQueue instance containing c’s elements. If c is a SortedSet or PriorityQueue instance, this priority queue will be ordered according to the same ordering. Otherwise, this priority queue will be ordered according to the natural ordering of its elements. This constructor throws ClassCastException when c’s elements cannot be compared to one another according to the priority queue’s ordering and NullPointerException when c or any of its elements contain the null reference.
- PriorityQueue(int initialCapacity) creates a PriorityQueue instance with the specified initialCapacity and which orders its elements according to their natural ordering. This constructor throws IllegalArgumentException when initialCapacity is less than 1.
- PriorityQueue(int initialCapacity, Comparator<? super E> comparator) creates a PriorityQueue instance with the specified initialCapacity and which orders its elements according to the specified comparator. Natural ordering is used when comparator contains the null reference. This constructor throws IllegalArgumentException when initialCapacity is less than 1.
CHAPTER 9: Exploring the Collections Framework

- PriorityQueue(PriorityQueue<? extends E> pq) creates a PriorityQueue instance containing pq’s elements. This priority queue will be ordered according to the same ordering as pq. This constructor throws ClassCastException when pq’s elements cannot be compared to one another according to pq’s ordering and NullPointerException when pq or any of its elements contains the null reference.

- PriorityQueue(SortedSet<? extends E> ss) creates a PriorityQueue instance containing ss’s elements. This priority queue will be ordered according to the same ordering as ss. This constructor throws ClassCastException when ss’s elements cannot be compared to one another according to ss’s ordering and NullPointerException when ss or any of its elements contains the null reference.

Listing 9-14 demonstrates a priority queue.

Listing 9-14. Adding Randomly Generated Integers to a Priority Queue

```java
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        Queue<Integer> qi = new PriorityQueue<Integer>();
        for (int i = 0; i < 15; i++)
            qi.add((int) (Math.random() * 100));
        while (!qi.isEmpty())
            System.out.print(qi.poll() + " ");
        System.out.println();
    }
}
```

After creating a priority queue, PriorityQueueDemo’s main thread adds 15 randomly generated integers (ranging from 0 through 99) to this queue. It then enters a while loop that repeatedly polls the priority queue for the next element and outputs that element until the queue is empty.

When you run this application, it outputs a line of 15 integers in ascending numerical order from left to right. For example, I observed the following output from one run:

```
30 43 53 61 66 66 67 76 78 80 83 87 90 97
```

Because poll() returns null when there are no more elements, I could have coded this loop as follows:

```java
Integer i;
while ((i = qi.poll()) != null)
    System.out.print(i + " ");
```
Suppose you want to reverse the order of the previous example's output so that the largest element appears on the left and the smallest element appears on the right. As Listing 9-15 demonstrates, you can achieve this task by passing a comparator to the appropriate PriorityQueue constructor.

Listing 9-15. Using a Comparator with a Priority Queue

```java
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
public class PriorityQueueDemo
{
    final static int NELEM = 15;

    public static void main(String[] args)
    {
        Comparator<Integer> cmp;
        cmp = new Comparator<Integer>()
        {
            @Override
            public int compare(Integer e1, Integer e2)
            {
                return e2 - e1;
            }
        };
        Queue<Integer> qi = new PriorityQueue<Integer>(NELEM, cmp);
        for (int i = 0; i < NELEM; i++)
            qi.add((int) (Math.random() * 100));
        while (!qi.isEmpty())
            System.out.print(qi.poll() + " ");
        System.out.println();
    }
}
```

Listing 9-15 is similar to Listing 9-14, but there are some differences. First, I have declared a constant named NELEM so that I can easily change both the priority queue's initial capacity and the number of elements inserted into the priority queue by specifying the new value in one place.

Second, Listing 9-15 declares and instantiates an anonymous class that implements Comparator. Its compareTo() method subtracts element e2 from element e1 to achieve descending numerical order. The compiler handles the task of unboxing e2 and e1 by converting e2 - e1 to e2.intValue() - e1.intValue().

Finally, Listing 9-15 passes an initial capacity of NELEM elements and the instantiated comparator to the PriorityQueue(int initialCapacity, Comparator<? super E> comparator) constructor. The priority queue will use this comparator to order these elements.

Run this application and you will now see a single output line of 15 integers shown in descending numerical order from left to right. For example, I observed this output line:

```
97 72 70 67 64 56 43 36 22 9 5 3 2 1
```
Exploring Deques

A deque (pronounced deck) is a double-ended queue in which element insertion or removal occurs at its head or tail. Deques can be used as queues or stacks.

Deque, whose generic type is Deque<E>, extends Queue in which the inherited add(E e) method inserts e at the deque’s tail. Table 9-6 describes Deque-specific methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addFirst(E e)</td>
<td>Inserts e at the head of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method offerFirst(). This method throws IllegalStateException when e cannot be added at this time because of capacity restrictions, ClassCastException when e’s class prevents e from being added to this deque, NullPointerException when e contains the null reference and this deque doesn’t permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this deque.</td>
</tr>
<tr>
<td>void addLast(E e)</td>
<td>Inserts e at the tail of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method offerLast(). This method throws IllegalStateException when e cannot be added at this time because of capacity restrictions, ClassCastException when e’s class prevents e from being added to this deque, NullPointerException when e contains the null reference and this deque doesn’t permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this deque.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; descendingIterator()</td>
<td>Returns an iterator over the elements in this deque in reverse sequential order. The elements will be returned in order from last (tail) to first (head). The inherited Iterator&lt;E&gt; iterator() method returns elements from the head to the tail.</td>
</tr>
<tr>
<td>E element()</td>
<td>Retrieves but doesn’t remove the first element of this deque (at the head). This method differs from peek() only in that it throws NoSuchElementException when this deque is empty. This method is equivalent to getFirst().</td>
</tr>
<tr>
<td>E getFirst()</td>
<td>Retrieves but doesn’t remove the first element of this deque. This method differs from peekFirst() only in that it throws NoSuchElementException when this deque is empty.</td>
</tr>
<tr>
<td>E getLast()</td>
<td>Retrieves but doesn’t remove the last element of this deque. This method differs from peekLast() only in that it throws NoSuchElementException when this deque is empty.</td>
</tr>
</tbody>
</table>

(continued)
CHAPTER 9: Exploring the Collections Framework

Table 9-6. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean offer(E e)</td>
<td>Inserts e at the tail of this deque if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false when no space is currently available. When using a capacity-restricted deque, this method is generally preferable to the add() method, which can fail to insert an element only by throwing an exception. This method throws ClassCastException when e’s class prevents e from being added to this deque, NullPointerException when e contains the null reference and this deque doesn’t permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this deque. This method is equivalent to offerLast().</td>
</tr>
<tr>
<td>boolean offerFirst(E e)</td>
<td>Inserts e at the head of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the addFirst() method, which can fail to insert an element only by throwing an exception. This method throws ClassCastException when e’s class prevents e from being added to this deque, NullPointerException when e contains the null reference and this deque doesn’t permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this deque.</td>
</tr>
<tr>
<td>boolean offerLast(E e)</td>
<td>Inserts e at the tail of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the addLast() method, which can fail to insert an element only by throwing an exception. This method throws ClassCastException when e’s class prevents e from being added to this deque, NullPointerException when e contains the null reference and this deque doesn’t permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this deque.</td>
</tr>
<tr>
<td>E peek()</td>
<td>Retrieves but doesn’t remove the first element of this deque (at the head), or returns null when this deque is empty. This method is equivalent to peekFirst().</td>
</tr>
<tr>
<td>E peekFirst()</td>
<td>Retrieves but doesn’t remove the first element of this deque (at the head), or returns null when this deque is empty.</td>
</tr>
<tr>
<td>E peekLast()</td>
<td>Retrieves but doesn’t remove the last element of this deque (at the tail), or returns null when this deque is empty.</td>
</tr>
<tr>
<td>E poll()</td>
<td>Retrieves and removes the first element of this deque (at the head), or returns null when this deque is empty. This method is equivalent to pollFirst().</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E pollFirst()</td>
<td>Retrieves and removes the first element of this deque (at the head), or returns null when this deque is empty.</td>
</tr>
<tr>
<td>E pollLast()</td>
<td>Retrieves and removes the last element of this deque (at the tail), or returns null when this deque is empty.</td>
</tr>
<tr>
<td>E pop()</td>
<td>Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque. This method is equivalent to removeFirst().</td>
</tr>
<tr>
<td>void push(E e)</td>
<td>Pushes e onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing IllegalStateException when no space is currently available. This method also throws ClassCastException when e’s class prevents e from being added to this deque, NullPointerException when e contains the null reference and this deque doesn’t permit null elements to be added, and IllegalArgumentException when some property of e prevents it from being added to this deque. This method is equivalent to addFirst().</td>
</tr>
<tr>
<td>E remove()</td>
<td>Retrieves and removes the first element of this deque (at the head). This method differs from poll() only in that it throws NoSuchElementException when this deque is empty. This method is equivalent to removeFirst().</td>
</tr>
<tr>
<td>E removeFirst()</td>
<td>Retrieves and removes the first element of this deque. This method differs from pollFirst() only in that it throws NoSuchElementException when this deque is empty.</td>
</tr>
<tr>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>removeFirstOccurrence(Object o)</td>
<td>Removes the first occurrence of o from this deque. If the deque doesn’t contain o, it is unchanged. Returns true when this deque contained o (or equivalently, when this deque changed as a result of the call). This method throws ClassCastException when o’s class prevents o from being added to this deque and NullPointerException when o contains the null reference and this deque doesn’t permit null elements to be added. The inherited boolean remove(Object o) method is equivalent to this method.</td>
</tr>
<tr>
<td>E removeLast()</td>
<td>Retrieves and removes the last element of this deque. This method differs from pollLast() only in that it throws NoSuchElementException when this deque is empty.</td>
</tr>
<tr>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>removeLastOccurrence(Object o)</td>
<td>Removes the last occurrence of o from this deque. If the deque doesn’t contain o, it is unchanged. Returns true when this deque contained o (or equivalently, when this deque changed as a result of the call). This method throws ClassCastException when o’s class prevents o from being added to this deque and NullPointerException when o contains the null reference and this deque doesn’t permit null elements to be added.</td>
</tr>
</tbody>
</table>
As Table 9-6 reveals, Deque declares methods to access elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception when the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Deque implementations; in most implementations, insert operations cannot fail.

Figure 9-2 reveals a table from Deque’s Java documentation that nicely summarizes both forms of the insert, remove, and examine methods for both the head and the tail.

![Figure 9-2](image)

**Figure 9-2.** Deque declares 12 methods for inserting, removing, and examining elements at the head or tail of a deque

When a deque is used as a queue, you observe FIFO behavior. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the Queue interface are precisely equivalent to the Deque methods as indicated in Table 9-7.

**Table 9-7. Queue and Equivalent Deque Methods**

<table>
<thead>
<tr>
<th>Queue Method</th>
<th>Equivalent Deque Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(e)</td>
<td>addLast(e)</td>
</tr>
<tr>
<td>offer(e)</td>
<td>offerLast(e)</td>
</tr>
<tr>
<td>remove()</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>poll()</td>
<td>pollFirst()</td>
</tr>
<tr>
<td>element()</td>
<td>getFirst()</td>
</tr>
<tr>
<td>peek()</td>
<td>peekFirst()</td>
</tr>
</tbody>
</table>

Finally, deques can also be used as LIFO stacks. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Because a stack’s push(e) method would be equivalent to Deque’s addFirst(e) method, its pop() method would be equivalent to Deque’s removeFirst() method, and its peek() method would be equivalent to Deque’s peekFirst() method, Deque declares the E peek(), E pop(), and void push(E e) stack-oriented convenience methods.
ArrayDeque

The ArrayDeque class provides a resizable-array implementation of the Deque interface. It prohibits null elements from being added to a deque, and its iterator() method returns fail-fast iterators.

ArrayDeque supplies three constructors:

- ArrayDeque() creates an empty array deque with an initial capacity of 16 elements.
- ArrayDeque(Collection<? extends E> c) creates an array deque containing c’s elements in the order in which they are returned by c’s iterator. (The first element returned by c’s iterator becomes the first element or front of the deque.) NullPointerException is thrown when c contains the null reference.
- ArrayDeque(int numElements) creates an empty array deque with an initial capacity sufficient to hold numElements elements. No exception is thrown when the argument passed to numElements is less than or equal to zero.

Listing 9-16 demonstrates an array deque.

Listing 9-16. Using an Array Deque as a Stack

```java
import java.util.ArrayDeque;
import java.util.Deque;
public class ArrayDequeDemo {
    public static void main(String[] args) {
        Deque<String> stack = new ArrayDeque<String>();
        String[] weekdays = { "Sunday", "Monday", "Tuesday", "Wednesday",
                            "Thursday", "Friday", "Saturday" };
        for (String weekday: weekdays)
            stack.push(weekday);
        while (stack.peek() != null)
            System.out.println(stack.pop());
    }
}
```

ArrayDequeDemo creates a deque for use as a stack and an array of weekday names. It then pushes these names on this stack and pops them, outputting the names in reverse order.

When you run this application, it generates the following output:

Saturday
Friday
Thursday
Wednesday
Tuesday
Monday
Sunday
Exploring Maps

A *map* is a group of key/value pairs (also known as *entries*). Because the *key* identifies an entry, a map cannot contain duplicate keys. Furthermore, each key can map to at most one value. Maps are described by the `Map` interface, which has no parent interface, and whose generic type is `Map<K,V>` (*K* is the key's type; *V* is the value's type).

Table 9-8 describes `Map`'s methods.

**Table 9-8. Map Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void clear()</code></td>
<td>Removes all elements from this map, leaving it empty. This method throws <code>UnsupportedOperationException</code> when <code>clear()</code> is not supported.</td>
</tr>
<tr>
<td><code>boolean containsKey(Object key)</code></td>
<td>Returns true when this map contains an entry for the specified key; otherwise, returns false. This method throws <code>ClassCastException</code> when key is of an inappropriate type for this map and <code>NullPointerException</code> when key contains the null reference and this map doesn't permit null keys.</td>
</tr>
<tr>
<td><code>boolean containsValue(Object value)</code></td>
<td>Returns true when this map maps one or more keys to value. This method throws <code>ClassCastException</code> when value is of an inappropriate type for this map and <code>NullPointerException</code> when value contains the null reference and this map doesn't permit null values.</td>
</tr>
<tr>
<td><code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code></td>
<td>Returns a Set view of the entries contained in this map. Because this map backs the view, changes that are made to the map are reflected in the set and vice versa.</td>
</tr>
<tr>
<td><code>boolean equals(Object o)</code></td>
<td>Compares o with this map for equality. Returns true when o is also a map and the two maps represent the same entries; otherwise, returns false.</td>
</tr>
<tr>
<td><code>V get(Object key)</code></td>
<td>Returns the value to which key is mapped or null when this map contains no entry for key. If this map permits null values, then a return value of null doesn't necessarily indicate that the map contains no entry for key; it is also possible that the map explicitly maps key to the null reference. The <code>containsKey()</code> method may be used to distinguish between these two cases. This method throws <code>ClassCastException</code> when key is of an inappropriate type for this map and <code>NullPointerException</code> when key contains the null reference and this map doesn't permit null keys.</td>
</tr>
<tr>
<td><code>int hashCode()</code></td>
<td>Returns the hash code for this map. A map's hash code is defined to be the sum of the hash codes for the entries in the map's <code>entrySet()</code> view.</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>Returns true when this map contains no entries; otherwise, returns false.</td>
</tr>
<tr>
<td><code>Set&lt;K&gt; keySet()</code></td>
<td>Returns a Set view of the keys contained in this map. Because this map backs the view, changes that are made to the map are reflected in the set and vice versa.</td>
</tr>
</tbody>
</table>

(continued)
Table 9-8. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>V put(K key, V value)</code></td>
<td>Associates value with key in this map. If the map previously contained an entry for key, the old value is replaced by value. This method returns the previous value associated with key or null when there was no entry for key. (The null return value can also indicate that the map previously associated the null reference with key, if the implementation supports null values.) This method throws <code>UnsupportedOperationException</code> when <code>put()</code> is not supported, <code>ClassCastException</code> when key’s or value’s class is not appropriate for this map, <code>IllegalArgumentException</code> when some property of key or value prevents it from being stored in this map, and <code>NullPointerException</code> when key or value contains the null reference and this map doesn’t permit null keys or values.</td>
</tr>
<tr>
<td><code>void putAll(Map&lt;? extends K, ? extends V&gt; m)</code></td>
<td>Copies all entries from map <code>m</code> to this map. The effect of this call is equivalent to that of calling <code>put(k, v)</code> on this map once for each mapping from key <code>k</code> to value <code>v</code> in map <code>m</code>. This method throws <code>UnsupportedOperationException</code> when <code>putAll()</code> is not supported, <code>ClassCastException</code> when the class of a key or value in map <code>m</code> is not appropriate for this map, <code>IllegalArgumentException</code> when some property of a key or value in map <code>m</code> prevents it from being stored in this map, and <code>NullPointerException</code> when <code>m</code> contains the null reference or when <code>m</code> contains null keys or values and this map doesn’t permit null keys or values.</td>
</tr>
<tr>
<td><code>V remove(Object key)</code></td>
<td>Removes key’s entry from this map when it is present. This method returns the value to which this map previously associated with key or null when the map contained no mapping for key. If this map permits null values, then a return value of null doesn’t necessarily indicate that the map contained no entry for key; it is also possible that the map explicitly mapped key to null. This map will not contain an entry for key once the call returns. This method throws <code>UnsupportedOperationException</code> when <code>remove()</code> is not supported, <code>ClassCastException</code> when the class of key is not appropriate for this map, and <code>NullPointerException</code> when key contains the null reference and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>Returns the number of key/value entries in this map. If the map contains more than <code>Integer.MAX_VALUE</code> entries, this method returns <code>Integer.MAX_VALUE</code>.</td>
</tr>
<tr>
<td><code>Collection&lt;V&gt; values()</code></td>
<td>Returns a <code>Collection</code> view of the values contained in this map. Because this map backs the view, changes that are made to the map are reflected in the collection and vice versa.</td>
</tr>
</tbody>
</table>
Unlike List, Set, and Queue, Map doesn't extend Collection. However, it is possible to view a map as a Collection instance by calling Map’s keySet(), values(), and entrySet() methods, which respectively return a Set of keys, a Collection of values, and a Set of key/value pair entries.

**Note** The values() method returns Collection instead of Set because multiple keys can map to the same value, and values() would then return multiple copies of the same value.

The Collection views returned by these methods (recall that a Set is a Collection because Set extends Collection) provide the only means to iterate over a Map. For example, suppose you declare Listing 9-17’s Color enum with its three Color constants, RED, GREEN, and BLUE.

### Listing 9-17. A Colorful enum

```java
class Color {
    RED(255, 0, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255);

    private int r, g, b;

    private Color(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    @Override
    public String toString() {
        return "r = " + r + ", g = " + g + ", b = " + b;
    }
}
```

The following example declares a map of String keys and Color values, adds several entries to the map, and iterates over the keys and values:

```java
Map<String, Color> colorMap = ...; // ... represents the creation of a Map implementation
colorMap.put("red", Color.RED);
colorMap.put("blue", Color.BLUE);
colorMap.put("green", Color.GREEN);
colorMap.put("RED", Color.RED);
for (String colorKey: colorMap.keySet())
    System.out.println(colorKey);
Collection<Color> colorValues = colorMap.values();
for (Iterator<Color> it = colorValues.iterator(); it.hasNext();)
    System.out.println(it.next());
```
When running this code fragment against a hashmap implementation (discussed later) of \texttt{colorMap},
you should observe output similar to the following:

\begin{verbatim}
red
blue
green
RED
r = 255, g = 0, b = 0
r = 0, g = 0, b = 255
r = 0, g = 255, b = 0
r = 255, g = 0, b = 0
\end{verbatim}

The first four output lines identify the map’s keys; the second four output lines identify the map’s values.

The \texttt{entrySet()} method returns a Set of \texttt{Map.Entry} objects. Each of these objects describes a single
entry as a key/value pair and is an instance of a class that implements the \texttt{Map.Entry} interface,
where \texttt{Entry} is a nested interface of \texttt{Map}. Table 9-9 describes \texttt{Map.Entry}’s methods.

\begin{table}[h]
\centering
\caption{\texttt{Map.Entry} Methods}
\begin{tabular}{ll}
\hline
Method & Description \\
\hline
\texttt{boolean equals(Object o)} & Compares \texttt{o} with this entry for equality. Returns true when \texttt{o} is also a map entry and the two entries have the same key and value. \\
\texttt{K getKey()} & Returns this entry’s key. This method optionally throws \texttt{IllegalStateException} when this entry has previously been removed from the backing map. \\
\texttt{V getValue()} & Returns this entry’s value. This method optionally throws \texttt{IllegalStateException} when this entry has previously been removed from the backing map. \\
\texttt{int hashCode()} & Returns this entry’s hash code. \\
\texttt{V setValue(V value)} & Replaces this entry’s value with \texttt{value}. The backing map is updated with the new value. This method throws \texttt{UnsupportedOperationException} when \texttt{setValue()} is not supported, \texttt{ClassCastException} when \texttt{value}’s class prevents it from being stored in the backing map, \texttt{NullPointerException} when \texttt{value} contains the null reference and the backing map doesn’t permit null, \texttt{IllegalArgumentException} when some property of \texttt{value} prevents it from being stored in the backing map, and (optionally) \texttt{IllegalStateException} when this entry has previously been removed from the backing map. \\
\hline
\end{tabular}
\end{table}

The following example shows you how you might iterate over the previous example’s map entries:

\begin{verbatim}
for (Map.Entry<String, Color> colorEntry: colorMap.entrySet())
    System.out.println(colorEntry.getKey() + ": " + colorEntry.getValue());
\end{verbatim}
When running this example against the previously mentioned hashmap implementation, you would observe the following output:

red: r = 255, g = 0, b = 0  
blue: r = 0, g = 0, b = 255  
green: r = 0, g = 255, b = 0  
RED: r = 255, g = 0, b = 0

**TreeMap**

The TreeMap class provides a map implementation that is based on a red-black tree. As a result, entries are stored in sorted order of their keys. However, accessing these entries is somewhat slower than with the other Map implementations (which are not sorted) because links must be traversed.


TreeMap supplies four constructors:

- TreeMap() creates a new, empty tree map that is sorted according to the natural ordering of its keys. All keys inserted into the map must implement the Comparable interface.
- TreeMap(Comparator<? super K> comparator) creates a new, empty tree map that is sorted according to the specified comparator. Passing null to comparator implies that natural ordering will be used.
- TreeMap(Map<? extends K, ? extends V> m) creates a new tree map containing m’s entries, sorted according to the natural ordering of its keys. All keys inserted into the new map must implement the Comparable interface. This constructor throws ClassCastException when m’s keys don’t implement Comparable or are not mutually comparable and NullPointerException when m contains the null reference.
- TreeMap(SortedMap<K, ? extends V> sm) creates a new tree map containing the same entries and using the same ordering as sm. (I discuss sorted maps later in this chapter.) This constructor throws NullPointerException when sm contains the null reference.

Listing 9-18 demonstrates a tree map.

**Listing 9-18. Sorting a Map’s Entries According to the Natural Ordering of Their String-Based Keys**

```java
import java.util.Map;
import java.util.TreeMap;

public class TreeMapDemo
{
    public static void main(String[] args)
```
TreeMapDemo creates a tree map and an array of fruit names. It then populates this map with these names and dumps the map's entries to standard output.

When you run this application, it generates the following output:

apples: 10
bananas: 17
grapes: 8
kiwis: 30
pears: 15

**HashMap**

The HashMap class provides a map implementation that is based on a hashtable data structure. This implementation supports all Map operations and permits null keys and null values. It makes no guarantees on the order in which entries are stored.

A hashtable maps keys to integer values with the help of a hash function. Java provides this function in the form of Object's hashCode() method, which classes override to provide appropriate hash codes.

A hash code identifies one of the hashtable’s array elements, which is known as a bucket or slot. For some hashtables, the bucket may store the value that is associated with the key. Figure 9-3 illustrates this kind of hashtable.
The hash function hashes Bob Doe to 0, which identifies the first bucket. This bucket contains ACCTS, which is Bob Doe’s employee type. The hash function also hashes John Doe and Sally Doe to 1 and 2 (respectively) whose buckets contain SALES.

A perfect hash function hashes each key to a unique integer value. However, this ideal is very difficult to meet. In practice, some keys will hash to the same integer value. This nonunique mapping is referred to as a collision.

To address collisions, most hashtables associate a linked list of entries with a bucket. Instead of containing a value, the bucket contains the address of the first node in the linked list, and each node contains one of the colliding entries. See Figure 9-4.

Figure 9-3. A simple hashtable maps keys to buckets that store values associated with those keys

Figure 9-4. A complex hashtable maps keys to buckets that store references to linked lists whose node values are hashed from the same keys
When storing a value in a hashtable, the hashtable uses the hash function to hash the key to its hash code, and then searches the appropriate linked list to see if an entry with a matching key exists. If there is an entry, its value is updated with the new value. Otherwise, a new node is created, populated with the key and value, and appended to the list.

When retrieving a value from a hashtable, the hashtable uses the hash function to hash the key to its hash code and then searches the appropriate linked list to see if an entry with a matching key exists. If there is an entry, its value is returned. Otherwise, the hashtable may return a special value to indicate that there is no entry, or it might throw an exception.

The number of buckets is known as the hashtable's capacity. The ratio of the number of stored entries divided by the number of buckets is known as the hashtable's load factor. Choosing the right load factor is important for balancing performance with memory use.

- As the load factor approaches 1, the probability of collisions and the cost of handling them (by searching lengthy linked lists) increase.
- As the load factor approaches 0, the hashtable's size in terms of number of buckets increases with little improvement in search cost.
- For many hashtables, a load factor of 0.75 is close to optimal. This value is the default for HashMap's hashtable implementation.

HashMap supplies four constructors:

- HashMap() creates a new, empty hashmap with an initial capacity of 16 and a load factor of 0.75.
- HashMap(int initialCapacity) creates a new, empty hashmap with a capacity specified by initialCapacity and a load factor of 0.75. This constructor throws IllegalArgumentException when initialCapacity's value is less than 0.
- HashMap(int initialCapacity, float loadFactor) creates a new, empty hashmap with a capacity specified by initialCapacity and a load factor specified by loadFactor. This constructor throws IllegalArgumentException when initialCapacity is less than 0 or when loadFactor is less than or equal to 0.
- HashMap(Map<? extends K, ? extends V> m) creates a new hashmap containing m's entries. This constructor throws NullPointerException when m contains the null reference.

Listing 9-19 demonstrates a hashmap.

Listing 9-19. Using a Hashmap to Count Command-Line Arguments

```java
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo
{
    public static void main(String[] args)
    {
        Map<String, Integer> argMap = new HashMap<String, Integer>();
        for (String arg: args)
```
{    Integer count = argMap.get(arg);    argMap.put(arg, (count == null) ? 1 : count + 1);}
System.out.println(argMap);
System.out.println("Number of distinct arguments = " + argMap.size());
}

HashMapDemo creates a hashmap of String keys and Integer values. Each key is one of the command-line arguments passed to this application, and its value is the number of occurrences of that argument on the command line.

For example, java HashMapDemo how much wood could a woodchuck chuck if a woodchuck could chuck wood generates the following output:

{wood=2, could=2, how=1, if=1, chuck=2, a=2, woodchuck=2, much=1}
Number of distinct arguments = 8

Note  LinkedHashMap is a subclass of HashMap that uses a linked list to store its entries. As a result, LinkedHashMap’s iterator returns entries in the order in which they were inserted. For example, if Listing 9-19 had specified Map<String, Integer> argMap = new LinkedHashMap<String, Integer>();, the application’s output for java HashMapDemo how much wood could a woodchuck chuck if a woodchuck could chuck wood would have been {how=1, much=1, wood=2, could=2, a=2, woodchuck=2, chuck=2, if=1} followed by Number of distinct arguments = 8.

Overriding hashCode()

Because the String class overrides equals() and hashCode(), Listing 9-19 can use String objects as keys in a hashmap. When you create a class whose instances are to be used as keys, you must ensure that you override both methods.

Listing 9-6 showed you that a class’s overriding hashCode() method can call a reference field’s hashCode() method and return its value, provided that the class declares a single reference field (and no primitive-type fields).

More commonly, classes declare multiple fields, and a better implementation of the hashCode() method is required. The implementation should try to generate hash codes that minimize collisions.

There is no rule on how to best implement hashCode(), and various algorithms (recipes for accomplishing tasks) have been created. My favorite algorithm appears in Effective Java, Second Edition, by Joshua Bloch (Addison-Wesley, 2008; ISBN: 0321356683).
The following algorithm, which assumes the existence of an arbitrary class that is referred to as $X$, closely follows Bloch’s algorithm, but is not identical:

1. Initialize int variable `hashCode` (the name is arbitrary) to an arbitrary nonzero integer value, such as 19. This variable is initialized to a nonzero value to ensure that it takes into account any initial fields whose hash codes are zeros. If you initialize `hashCode` to 0, the final hash code will be unaffected by such fields and you run the risk of increased collisions.

2. For each field $f$ that is also used in $X$’s `equals()` method, calculate $f$’s hash code and assign it to `int` variable `hc` as follows:
   a. If $f$ is of Boolean type, calculate $hc = f ? 1 : 0$.
   b. If $f$ is of byte integer, character, integer, or short integer type, calculate $hc = (int) f$. The integer value is the hash code.
   c. If $f$ is of long integer type, calculate $hc = (int) (f ^ (f >>> 32))$. This expression exclusive ORs the long integer’s least significant 32 bits with its most significant 32 bits.
   d. If $f$ is of type floating-point, calculate $hc = Float.floatToIntBits(f)$. This method takes $+\infty$, $-\infty$, and NaN into account.
   e. If $f$ is of type double precision floating-point, calculate $hc = Double.doubleToLongBits(f); hc = (int) (l ^ (l >>> 32))$.
   f. If $f$ is a reference field with a null reference, calculate $hc = 0$.
   g. If $f$ is a reference field with a nonnull reference, and if $X$’s `equals()` method compares the field by recursively calling `equals()` (as in Listing 9-12’s `Employee` class), calculate $hc = f.hashCode()$. However, if `equals()` employs a more complex comparison, create a canonical (simplest possible) representation of the field and call `hashCode()` on this representation.
   h. If $f$ is an array, treat each element as a separate field by applying this algorithm recursively and combining the `hc` values as shown in the next step.

3. Combine `hc` with `hashCode` as follows: `hashCode = hashCode * 31 + hc`. Multiplying `hashCode` by 31 makes the resulting hash value dependent on the order in which fields appear in the class, which improves the hash value when a class contains multiple fields that are similar (several ints, for example). I chose 31 to be consistent with the `String` class’s `hashCode()` method.


In Chapter 4, Listing 4–7’s `Point` class overrode `equals()` but didn’t override `hashCode()`. I later presented a small code fragment that must be appended to `Point`’s `main()` method to demonstrate the problem of not overriding `hashCode()`. I restate this problem here:

*Although objects $p1$ and `Point(10, 20)` are logically equivalent, these objects have different hash codes, resulting in each object referring to a different entry in the hashmap. If an object is not stored (via `put()`) in that entry, `get()` returns null.*

Listing 9-20 modifies Listing 4–7’s `Point` class by declaring a `hashCode()` method. This method uses the aforementioned algorithm to ensure that logically equivalent `Point` objects hash to the same entry.
Listing 9-20. Overriding hashCode() to Return Proper Hash Codes for Point Objects

```java
import java.util.HashMap;
import java.util.Map;

public class Point {
    private int x, y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }

    @Override
    public int hashCode() {
        int hashCode = 19;
        int hc = x;
        hashCode = hashCode * 31 + hc;
        hc = y;
        hashCode = hashCode * 31 + hc;
        return hashCode;
    }

    public static void main(String[] args) {
        Point p1 = new Point(10, 20);
        Point p2 = new Point(20, 30);
        Point p3 = new Point(10, 20);
    }
```
// Test reflexivity
System.out.println(p1.equals(p1)); // Output: true

// Test symmetry
System.out.println(p1.equals(p2)); // Output: false
System.out.println(p2.equals(p1)); // Output: false

// Test transitivity
System.out.println(p2.equals(p3)); // Output: false
System.out.println(p1.equals(p3)); // Output: true

// Test nullability
System.out.println(p1.equals(null)); // Output: false

// Extra test to further prove the instanceof operator's usefulness.
System.out.println(p1.equals("abc")); // Output: false

Map<Point, String> map = new HashMap<Point, String>();
map.put(p1, "first point");
System.out.println(map.get(p1)); // Output: first point
System.out.println(map.get(new Point(10, 20))); // Output: first point

Listing 9-20's `hashCode()` method is a little verbose in that it assigns each of x and y to local variable `hc` rather than directly using these fields in the hash code calculation. However, I decided to follow this approach to more closely mirror the hash code algorithm.

When you run this application, its last two lines of output are of the most interest. Instead of presenting `first point` followed by `null` on two separate lines, the application now correctly presents `first point` followed by `first point` on these lines.

**HashMap and Image Caches**

In Chapter 8, I introduced the `java.lang.ref.SoftReference` class and stated that this class is useful for implementing caches of objects. One kind of cache is an *image cache*, which keeps images in memory (because it takes time to load them from disk) and ensures that duplicate (and possibly very large) images are not stored in memory, which helps to avoid out-of-memory errors.

The image cache contains references to image objects that are already in memory. If these references were strong, the images would remain in memory. You would then need to figure out which images are no longer needed and remove them from memory so that they can be garbage collected.

Having to manually remove images duplicates the work of a garbage collector. However, when you wrap the references to the image objects in `SoftReference` objects, the garbage collector will determine when to remove these objects (typically when heap memory runs low) and perform the removal on your behalf.

Listing 9-21 presents a generic `SoftCache` class that combines `SoftReference` with the `HashMap` class to implement a cache of images (or another kind of objects).
Listing 9-21. A Generic Class for Caching Arbitrary Objects in a HashMap

```java
import java.lang.ref.SoftReference;
import java.util.HashMap;

public class SoftCache<K, V> {
    private HashMap<K, SoftReference<V>> map;

    public SoftCache() {
        map = new HashMap<K, SoftReference<V>>();
    }

    public V get(K key) {
        SoftReference<V> softRef = map.get(key);
        if (softRef == null)
            return null;
        return softRef.get();
    }

    public V put(K key, V value) {
        SoftReference<V> softRef = map.put(key, new SoftReference<V>(value));
        if (softRef == null)
            return null;
        V oldValue = softRef.get();
        softRef.clear();
        return oldValue;
    }

    public V remove(K key) {
        SoftReference<V> softRef = map.remove(key);
        if (softRef == null)
            return null;
        V oldValue = softRef.get();
        softRef.clear();
        return oldValue;
    }
}
```

Listing 9-21’s SoftCache class is pretty straightforward. It declares a private HashMap field and initializes this field in its constructor. It also provides get(), put(), and remove() methods for interacting with the cache.

One item that might be confusing is the softRef.clear() method call in each of the put() and remove() methods. This call makes the previously stored referent (whose SoftReference container instance is being overwritten, in the case of put(), or removed, in the case of remove()), which is being returned from put() or remove(), eligible for cleanup. However, the value will not be cleaned
up when it's assigned to a variable, which would provide a strong reference to the value, and so it must be cleared.

Listing 9-22 presents an application that uses a SoftCache instance to cache (hypothetical) images, retrieve cached images before (hypothetically) drawing them, and re-cache images that are no longer cached.

**Listing 9-22. Caching Images**

import java.lang.ref.SoftReference;

class Image {
    private byte[] image;

    private Image(String name) {
        image = new byte[1024 * 1024 * 100];
    }

    static Image getImage(String name) {
        return new Image(name);
    }
}

public class SoftCacheDemo {
    public static void main(String[] args) {
        SoftCache<Integer, Image> sc = new SoftCache<Integer, Image>();
        int i = 0;
        while (true) {
            System.out.printf("Putting large image %d into soft cache%n", i);
            sc.put(i, Image.getImage("large.png" + i));
            i++;
            int x = (int) (Math.random() * i);
            System.out.printf("Acquiring image %d from cache.%n", x);
            Image im = sc.get(x);
            if (im == null) {
                System.out.printf("Image %d no longer in cache. Re-caching.%n", x);
                sc.put(x, im = Image.getImage("large.png" + x));
            }
            System.out.printf("Drawing image %d%n", x);
            im = null; // Remove strong reference to image.
        }
    }
}
Listing 9-22 declares an `Image` class that simulates the task of loading a large image, and declares a `SoftCacheDemo` class that demonstrates a `SoftCache` of `Image` objects.

The `main()` method first instantiates `SoftCache`. It then enters an infinite loop to continually cache new image objects and access image objects at random.

`SoftCache`'s `get()` method returns null when the image is no longer cached (or when no such image was ever put into the cache, which won’t happen in this application). At this point, the image is re-obtained and re-cached via `SoftCache`'s `put()` method.

Compile Listing 9-22 (`javac SoftCacheDemo.java`) and run the application (`java SoftCacheDemo`). You should discover output that’s similar to the following partial output (on a Windows 7 platform):

```plaintext
Acquiring image 6 from cache.
Image 6 no longer in cache. Re-caching.
Drawing image 6.
Putting large image 34 into soft cache.
Acquiring image 34 from cache.
Drawing image 34.
Putting large image 35 into soft cache.
Acquiring image 7 from cache.
Image 7 no longer in cache. Re-caching.
Drawing image 7.
Putting large image 36 into soft cache.
Acquiring image 1 from cache.
Image 1 no longer in cache. Re-caching.
Drawing image 1.
Putting large image 37 into soft cache.
Acquiring image 1 from cache.
Drawing image 1.
Putting large image 38 into soft cache.
Acquiring image 0 from cache.
Image 0 no longer in cache. Re-caching.
Drawing image 0.
Putting large image 39 into soft cache.
Acquiring image 34 from cache.
Image 34 no longer in cache. Re-caching.
Drawing image 34.
Putting large image 40 into soft cache.
Acquiring image 14 from cache.
Image 14 no longer in cache. Re-caching.
Drawing image 14.
Putting large image 41 into soft cache.
Acquiring image 38 from cache.
Drawing image 38.
Putting large image 42 into soft cache.
Acquiring image 39 from cache.
Drawing image 39.
Putting large image 43 into soft cache.
Acquiring image 27 from cache.
Image 27 no longer in cache. Re-caching.
Drawing image 27.
```
IdentityHashMap

The IdentityHashMap class provides a Map implementation that uses reference equality (==) instead of object equality (equals()) when comparing keys and values. This is an intentional violation of Map’s general contract, which mandates the use of equals() when comparing elements.

IdentityHashMap obtains hash codes via System’s int identityHashCode(Object x) class method instead of via each key’s hashCode() method. identityHashCode() returns the same hash code for x as returned by Object’s hashCode() method, whether or not x’s class overrides hashCode(). The hash code for the null reference is zero.

These characteristics give IdentityHashMap a performance advantage over other Map implementations. Also, IdentityHashMap supports mutable keys (objects used as keys and whose hash codes change when their field values change while in the map). Listing 9-23 contrasts IdentityHashMap with HashMap where mutable keys are concerned.

Listing 9-23. Contrasting IdentityHashMap with HashMap in a Mutable Key Context

```java
import java.util.IdentityHashMap;
import java.util.HashMap;
import java.util.Map;

public class IdentityHashMapDemo {
    public static void main(String[] args) {
        Map<Employee, String> map1 = new IdentityHashMap<Employee, String>();
        Map<Employee, String> map2 = new HashMap<Employee, String>();
        Employee e1 = new Employee("John Doe", 28);
        map1.put(e1, "SALES");
        System.out.println(map1);
        Employee e2 = new Employee("Jane Doe", 26);
        map2.put(e2, "MGMT");
        System.out.println(map2);
        System.out.println("map1 contains key e1 = " + map1.containsKey(e1));
        System.out.println("map2 contains key e2 = " + map2.containsKey(e2));
        e1.setAge(29);
        e2.setAge(27);
        System.out.println(map1);
        System.out.println(map2);
        System.out.println("map1 contains key e1 = " + map1.containsKey(e1));
        System.out.println("map2 contains key e2 = " + map2.containsKey(e2));
    }
}
```
class Employee
{
    private String name;
    private int age;

    Employee(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    @Override
    public boolean equals(Object o)
    {
        if (!(o instanceof Employee))
            return false;
        Employee e = (Employee) o;
        return e.name.equals(name) && e.age == age;
    }

    @Override
    public int hashCode()
    {
        int hashCode = 19;
        hashCode = hashCode * 31 + name.hashCode();
        hashCode = hashCode * 31 + age;
        return hashCode;
    }

    void setAge(int age)
    {
        this.age = age;
    }

    void setName(String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return name + " " + age;
    }
}
Listing 9-23’s main() method creates IdentityHashMap and HashMap instances that each store an entry consisting of an Employee key and a String value. Because Employee instances are mutable (because of setAge() and setName()), main() changes their ages while these keys are stored in their maps. These changes result in the following output:

{John Doe 28=SALES}
{Jane Doe 26=MGMT}
map1 contains key e1 = true
map2 contains key e2 = true
{John Doe 29=SALES}
{Jane Doe 27=MGMT}
map1 contains key e1 = true
map2 contains key e2 = false

The last four lines show that the changed entries remain in their maps. However, map2’s containsKey() method reports that its HashMap instance no longer contains its Employee key (which should be Jane Doe 27), whereas map1’s containsKey() method reports that its IdentityHashMap instance still contains its Employee key, which is now John Doe 29.

Note IdentityHashMap’s documentation states that “a typical use of this class is topology-preserving object graph transformations, such as serialization or deep copying.” (I discuss serialization in Chapter 11.) It also states that “another typical use of this class is to maintain proxy objects.” Also, stackoverflow’s “Use Cases for Identity HashMap” topic ([http://stackoverflow.com/questions/838528/use-cases-for-identity-hashmap](http://stackoverflow.com/questions/838528/use-cases-for-identity-hashmap)) mentions that it is much faster to use IdentityHashMap than HashMap when the keys are java.lang.Class objects.

**WeakHashMap**

The WeakHashMap class provides a Map implementation that’s based on weakly reachable keys. Each key object is stored indirectly as the referent of a weak reference; and an entry is automatically removed from the map after the garbage collector clears all weak references to the entry’s key.

Note Check out Chapter 8’s “Exploring References” section to learn about weakly reachable and weak references.

WeakHashMap declares the same four constructors as HashMap. Listing 9-24 uses its noargument constructor to initialize a weak hashmap and detects when an entry is removed.
Listing 9-24. Detecting a Weak Hashmap Entry’s Removal

```java
import java.util.Map;
import java.util.WeakHashMap;

class LargeObject
{
    private byte[] memory = new byte[1024 * 1024 * 50]; // 50 megabytes
}

class WeakHashMapDemo
{
    public static void main(String[] args)
    {
        Map<LargeObject, String> map = new WeakHashMap<>();
        LargeObject lo = new LargeObject();
        map.put(lo, "Large Object");
        System.out.println(map);
        lo = null;
        while (!map.isEmpty())
        {
            System.out.println("looping until map is empty");
            new LargeObject();
        }
        System.out.println(map);
    }
}
```

Listing 9-24’s main() method stores a 50MB LargeObject key and a String value in the weak hashmap and then removes the key’s strong reference by assigning null to lo. main() next enters a while loop that executes until the map is empty (map.isEmpty() returns true).

The loop first outputs a message. It then creates a LargeObject object, throwing away its reference. This activity should eventually cause the garbage collector to run and remove the map’s solitary entry. However, if you discover an unending loop, insert System.gc(); into this loop. This method call hints to the virtual machine that now might be a good time to run the garbage collector.

Compile Listing 9-24 (javac WeakReferenceDemo.java) and run the application (java WeakReferenceDemo). You should discover output that’s similar to the following Windows 7 output:

```
{LargeObject@6cc2060e=Large Object}
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
looping until map is empty
```
EnumMap

The EnumMap class provides a Map implementation whose keys are the members of the same enum. Null keys are not permitted; any attempt to store a null key results in a thrown NullPointerException. Because an enum map is represented internally as an array, an enum map approaches an array in terms of performance.

EnumMap supplies the following constructors:

- EnumMap(Class<K> keyType) creates an empty enum map with the specified keyType. This constructor throws NullPointerException when keyType contains the null reference.

- EnumMap(EnumMap<K,? extends V> m) creates an enum map with the same key type as m, and with m's entries. This constructor throws NullPointerException when m contains the null reference.

- EnumMap(Map<K,? extends V> m) creates an enum map initialized with m's entries. If m is an EnumMap instance, this constructor behaves like the previous constructor. Otherwise, m must contain at least one entry in order to determine the new enum map's key type. This constructor throws NullPointerException when m contains the null reference and IllegalArgumentException when m is not an EnumMap instance and is empty.

Listing 9-25 demonstrates EnumMap.

Listing 9-25. An Enum Map of Coin Constants

import java.util.EnumMap;
import java.util.Map;

class Coin
{
    PENNY, NICKEL, DIME, QUARTER
}
public class EnumMapDemo
{
    public static void main(String[] args)
    {
        Map<Coin, Integer> map = new EnumMap<Coin, Integer>(Coin.class);
        map.put(Coin.PENNY, 1);
        map.put(Coin.NICKEL, 5);
        map.put(Coin.DIME, 10);
        map.put(Coin.QUARTER, 25);
        System.out.println(map);
        Map<Coin, Integer> mapCopy = new EnumMap<Coin, Integer>(map);
        System.out.println(mapCopy);
    }
}

EnumMapDemo creates a map of Coin keys and Integer values. It then inserts several Coin instances into this map and outputs the map. Finally, it creates a copy of this map and outputs the copy.

When you run this application, it generates the following output:

{PENNY=1, NICKEL=5, DIME=10, QUARTER=25}
{PENNY=1, NICKEL=5, DIME=10, QUARTER=25}

Exploring Sorted Maps

TreeMap is an example of a sorted map, which is a map that maintains its entries in ascending order, sorted according to the keys’ natural ordering or according to a comparator that is supplied when the sorted map is created. Sorted maps are described by the SortedMap interface.

SortedMap (whose generic type is SortedMap<K, V>) extends Map. With two exceptions, the methods it inherits from Map behave identically on sorted maps as on other maps:

- The Iterator instance returned by the iterator() method on any of the sorted map’s Collection views traverses the collections in order.
- The arrays returned by the Collection views’ toArray() methods contain the keys, values, or entries in order.

Note Although not guaranteed, the toString() methods of the Collection views of SortedMap implementations in the Collections Framework (such as TreeMap) return a string containing all of the view’s elements in order.

SortedMap’s documentation requires that an implementation must provide the four standard constructors that I presented in my discussion of TreeMap. Furthermore, implementations of this interface must implement the methods that are described in Table 9-10.
### Table 9-10. SortedMap-Specific Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparator&lt;? super K&gt; comparator()</td>
<td>Returns the comparator used to order the keys in this map, or null when this map uses the natural ordering of its keys.</td>
</tr>
<tr>
<td>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</td>
<td>Returns a Set view of the mappings contained in this map. The set’s iterator returns these entries in ascending key order. Because the view is backed by this map, changes that are made to the map are reflected in the set and vice versa.</td>
</tr>
<tr>
<td>K firstKey()</td>
<td>Returns the first (lowest) key currently in this map, or throws a NoSuchElementException instance when this map is empty.</td>
</tr>
<tr>
<td>SortedMap&lt;K, V&gt; headMap(K toKey)</td>
<td>Returns a view of that portion of this map whose keys are strictly less than toKey. Because this map backs the returned map, changes in the returned map are reflected in this map and vice versa. The returned map supports all optional map operations that this map supports. This method throws ClassCastException when toKey is not compatible with this map’s comparator (or, when the map has no comparator, when toKey doesn’t implement Comparable), NullPointerException when toKey is null and this map doesn't permit null keys, and IllegalArgumentException when this map has a restricted range and toKey lies outside of this range’s bounds.</td>
</tr>
<tr>
<td>Set&lt;K&gt; keySet()</td>
<td>Returns a Set view of the keys contained in this map. The set’s iterator returns the keys in ascending order. Because the map backs the view, changes that are made to the map are reflected in the set and vice versa.</td>
</tr>
<tr>
<td>K lastKey()</td>
<td>Returns the last (highest) key currently in this map, or throws a NoSuchElementException instance when this map is empty.</td>
</tr>
<tr>
<td>SortedMap&lt;K, V&gt; subMap(K fromKey, K toKey)</td>
<td>Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive. (When fromKey and toKey are equal, the returned map is empty.) Because this map backs the returned map, changes in the returned map are reflected in this map and vice versa. The returned map supports all optional map operations that this map supports. This method throws ClassCastException when fromKey and toKey cannot be compared to one another using this map’s comparator (or, when the map has no comparator, using natural ordering), NullPointerException when fromKey or toKey is null and this map doesn't permit null keys, and IllegalArgumentException when fromKey is greater than toKey or when this map has a restricted range and fromKey or toKey lies outside of this range’s bounds.</td>
</tr>
</tbody>
</table>

(continued)
Listing 9-26 demonstrates a sorted map based on a tree map.


```java
import java.util.Comparator;
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapDemo
{
    public static void main(String[] args)
    {
        SortedMap<String, Integer> smsi = new TreeMap<String, Integer>();
        String[] officeSupplies = {
            "pen", "pencil", "legal pad", "CD", "paper"
        };
        int[] quantities = {
            20, 30, 5, 10, 20
        };
        for (int i = 0; i < officeSupplies.length; i++)
            smsi.put(officeSupplies[i], quantities[i]);
        System.out.println(smsi);
        System.out.println(smsi.headMap("pencil"));
        System.out.println(smsi.headMap("paper"));
        SortedMap<String, Integer> smsiCopy;
    }
}
```

Table 9-10. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortedMap&lt;K, V&gt; tailMap(K fromKey)</td>
<td>Returns a view of that portion of this map whose keys are greater than or equal to fromKey. Because this map backs the returned map, changes in the returned map are reflected in this map and vice versa. The returned map supports all optional map operations that this map supports. This method throws ClassCastException when fromKey is not compatible with this map’s comparator (or, when the map has no comparator, when fromKey doesn’t implement Comparable), NullPointerException when fromKey is null and this map doesn’t permit null elements, and IllegalArgumentException when this map has a restricted range and fromKey lies outside of the range’s bounds.</td>
</tr>
<tr>
<td>Collection&lt;V&gt; values()</td>
<td>Returns a Collection view of the values contained in this map. The collection’s iterator returns the values in ascending order of the corresponding keys. Because the map backs the collection, changes that are made to the map are reflected in the collection and vice versa.</td>
</tr>
</tbody>
</table>
SortedMapDemo creates a sorted map and arrays of office supply names and quantities. It then proceeds to populate the map from these arrays. After dumping out the map's contents and head views of parts of the map, it creates and outputs a copy of the map in descending order.

When you run this application, it generates the following output:

```
{CD=10, legal pad=5, paper=20, pen=20, pencil=30}
{CD=10, legal pad=5, paper=20, pen=20}
{CD=10, legal pad=5}
{pencil=30, pen=20, paper=20, legal pad=5, CD=10}
```

**Exploring Navigable Maps**

TreeMap is an example of a * navigable map, which is a sorted map that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets. Navigable maps are described by the NavigableMap interface, whose generic type is NavigableMap<K,V>, which extends SortedMap, and which is described in Table 9-11.
Table 9-11. NavigableMap-Specific Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map.Entry&lt;K,V&gt; ceilingEntry(K key)</td>
<td>Returns the key-value mapping associated with the least key greater than or equal to key, or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>K ceilingKey(K key)</td>
<td>Returns the least key greater than or equal to key, or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map, and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>NavigableSet&lt;K&gt; descendingKeySet()</td>
<td>Returns a reverse order navigable set-based view of the keys contained in this map. The set's iterator returns the keys in descending order. This map backs the set, so changes to the map are reflected in the set and vice versa. If the map is modified (except through the iterator's own remove() operation) while iterating over the set, the results of the iteration are undefined.</td>
</tr>
<tr>
<td>NavigableMap&lt;K,V&gt; descendingMap()</td>
<td>Returns a reverse order view of the mappings contained in this map. This map backs the descending map, so changes to the map are reflected in the descending map and vice versa. If either map is modified while iterating over a collection view of either map (except through the iterator's own remove() operation), the results of the iteration are undefined.</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; firstEntry()</td>
<td>Returns a key-value mapping associated with the least key in this map or null when the map is empty.</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; floorEntry(K key)</td>
<td>Returns a key-value mapping associated with the greatest key less than or equal to key or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>K floorKey(K key)</td>
<td>Returns the greatest key less than or equal to key or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>NavigableMap&lt;K,V&gt; headMap(K toKey, boolean inclusive)</td>
<td>Returns a view of the portion of this map whose keys are less than (or equal to, when inclusive is true) toKey. This map backs the returned map, so changes in the returned map are reflected in this map and vice versa. The returned map supports all optional map operations that this map supports. This method throws ClassCastException when toKey is not compatible with this map’s comparator (or, when the map has no comparator, when toMap doesn’t implement Comparable), NullPointerException when toKey is null and this map doesn’t permit null keys, and IllegalArgumentException when this map has a restricted range and toKey lies outside of this range’s bounds.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; higherEntry(K key)</td>
<td>Returns a key-value mapping associated with the least key strictly greater than key, or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>K higherKey(K key)</td>
<td>Returns the least key strictly greater than key, or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; lastEntry()</td>
<td>Returns a key-value mapping associated with the greatest key in this map, or null when the map is empty.</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; lowerEntry(K key)</td>
<td>Returns a key-value mapping associated with the greatest key strictly less than key, or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>K lowerKey(K key)</td>
<td>Returns the greatest key strictly less than key, or null when there is no such key. This method throws ClassCastException when key cannot be compared with the keys currently in the map and NullPointerException when key is null and this map doesn’t permit null keys.</td>
</tr>
<tr>
<td>NavigableSet&lt;K&gt; navigableKeySet()</td>
<td>Returns a navigable set-based view of the keys contained in this map. The set’s iterator returns the keys in ascending order. This map backs the set, so changes to the map are reflected in the set and vice versa. If the map is modified while iterating over the set (except through the iterator’s own remove() operation), the results of the iteration are undefined.</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; pollFirstEntry()</td>
<td>Removes and returns a key-value mapping associated with the least key in this map, or null when the map is empty.</td>
</tr>
<tr>
<td>Map.Entry&lt;K,V&gt; pollLastEntry()</td>
<td>Removes and returns a key-value mapping associated with the greatest key in this map, or null when the map is empty.</td>
</tr>
<tr>
<td>NavigableMap&lt;K,V&gt; subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)</td>
<td>Returns a view of the portion of this map whose keys range from fromKey to toKey. (When fromKey and toKey are equal, the returned map is empty unless fromInclusive and toInclusive are both true.) This map backs the returned map, so changes in the returned map are reflected in this map and vice versa. The returned map supports all optional map operations that this map supports. This method throws ClassCastException when fromKey and toKey cannot be compared to one another using this map’s comparator (or, when the map has no comparator, using natural ordering), NullPointerException when fromKey or toKey is null and this map doesn’t permit null elements, and IllegalArgumentException when fromKey is greater than toKey or when this map has a restricted range and fromKey or toKey lies outside of this range’s bounds.</td>
</tr>
</tbody>
</table>
Table 9-11. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NavigableMap&lt;K,V&gt;</td>
<td>Returns a view of the portion of this map whose keys are greater than (or equal to, when inclusive is true) fromKey. This map backs the returned map, so changes in the returned map are reflected in this map and vice versa. The returned map supports all optional map operations that this map supports. This method throws ClassCastException when fromKey is not compatible with this map's comparator (or, when the map has no comparator, when fromKey doesn't implement Comparable), NullPointerException when fromKey is null and this map doesn't permit null keys, and IllegalArgumentException when this map has a restricted range and fromKey lies outside of this range's bounds.</td>
</tr>
</tbody>
</table>

Table 9-11’s methods describe the NavigableMap equivalents of the NavigableSet methods presented in Table 9-4 and even return NavigableSet instances in two instances.

Listing 9-27 demonstrates a navigable map based on a tree map.

**Listing 9-27. Navigating a Map of (Bird, Count within A Small Acreage) Entries**

```java
import java.util.Iterator;
import java.util.NavigableMap;
import java.util.NavigableSet;
import java.util.TreeMap;

public class NavigableMapDemo
{
    public static void main(String[] args)
    {
        NavigableMap<String, Integer> nm = new TreeMap<String, Integer>();
        String[] birds = { "sparrow", "bluejay", "robin" };
        int[] ints = { 83, 12, 19 };
        for (int i = 0; i < birds.length; i++)
            nm.put(birds[i], ints[i]);
        System.out.println("Map = " + nm);
        System.out.print("Ascending order of keys: ");
        NavigableSet<String> ns = nm.navigableKeySet();
        Iterator iter = ns.iterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");
        System.out.println();
        System.out.print("Descending order of keys: ");
        ns = nm.descendingKeySet();
        iter = ns.iterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");
        System.out.println();
        System.out.println("First entry = " + nm.firstEntry());
        System.out.println("Last entry = " + nm.lastEntry());
        System.out.println("Entry < ostrich is " + nm.lowerEntry("ostrich"));
    }
}
```

www.it-ebooks.info
System.out.println("Entry > crow is " + nm.higherEntry("crow"));
System.out.println("Poll first entry: " + nm.pollFirstEntry());
System.out.println("Map = " + nm);
System.out.println("Poll last entry: " + nm.pollLastEntry());
System.out.println("Map = " + nm);

Listing 9-27's System.out.println("Map = " + nm); method calls rely on TreeMap's toString() method to obtain the contents of a navigable map.

When you run this application, you will observe the following output:

Map = {bluejay=12, robin=19, sparrow=83}
Ascending order of keys: bluejay robin sparrow
Descending order of keys: sparrow robin bluejay
First entry = bluejay=12
Last entry = sparrow=83
Entry < ostrich is bluejay=12
Entry > crow is robin=19
Poll first entry: bluejay=12
Map = {robin=19, sparrow=83}
Poll last entry: sparrow=83
Map = {robin=19}

Exploring the Arrays and Collections Utility APIs

The Collections Framework would be incomplete without its Arrays and Collections utility classes. Each class supplies various class methods that implement useful algorithms in the contexts of collections and arrays.

The following is a sampling of the Arrays class’s array-oriented utility methods:

- static <T> List<T> asList(T... a) returns a fixed-size list backed by array a. (Changes to the returned list “write through” to the array.) For example, List<String> birds = Arrays.asList("Robin", "Oriole", "Bluejay"); converts the three-element array of Strings (recall that a variable sequence of arguments is implemented as an array) to a List whose reference is assigned to birds.

- static int binarySearch(int[] a, int key) searches array a for entry key using the Binary Search algorithm (explained following this list). The array must be sorted before calling this method; otherwise, the results are undefined. This method returns the index of the search key, if it is contained in the array; otherwise, (-(insertion point) - 1) is returned. The insertion point is the point at which key would be inserted into the array (the index of the first element greater than key, or a.length if all elements in the array are less than key) and guarantees that the return value will be greater than or equal to 0 if and only if key is found. For example, Arrays.binarySearch(new String[] {"Robin", "Oriole", "Bluejay"}, "Oriole") returns 1, "Oriole"’s index.
static void fill(char[] a, char ch) stores ch in each element of the specified character array. For example, Arrays.fill(screen[i], ' '); fills the ith row of a 2D screen array with spaces.

static void sort(long[] a) sorts the elements in the long integer array a into ascending numerical order, for example, long lArray = new long[]{20000L, 89L, 66L, 33L}; Arrays.sort(lArray);

static <T> void sort(T[] a, Comparator<? super T> c) sorts the elements in array a using comparator c to order them. For example, when given
Comparator<String> cmp = new Comparator<String>() {
    @Override public int compare(String e1, String e2) {
        return e2.compareTo(e1);
    }
};
String[] innerPlanets = {"Mercury", "Venus", "Earth", "Mars"};
Arrays.sort(innerPlanets, cmp); uses cmp to help in sorting innerPlanets into descending order of its elements: Venus, Mercury, Mars, Earth is the result.

There are two common algorithms for searching an array for a specific element. Linear Search searches the array element by element from index 0 to the index of the searched-for element or the end of the array. On average, half of the elements must be searched; larger arrays take longer to search. However, the arrays don't need to be sorted.

In contrast, Binary Search searches ordered array a's n items for element e in a much faster amount of time. It works by recursively performing the following steps:

1. Set low index to 0.
2. Set high index to n - 1.
3. If low index > high index, then Print “Unable to find ” e. End.
4. Set e to a[middle index].
5. Set middle index to (low index + high index) / 2.
6. If e > a[middle index], then set low index to middle index + 1. Go to 3.
7. If e < a[middle index], then set high index to middle index - 1. Go to 3.
8. Print “Found ” e “at index ” middle index.

The algorithm is similar to optimally looking for a name in a phone book. Start by opening the book to the exact middle. If the name is not on that page, proceed to open the book to the exact middle of the first half or the second half, depending on which half the name occurs in. Repeat until you find the name (or not).

Applying a linear search to 4,000,000,000 elements results in approximately 2,000,000,000 comparisons (on average), which takes time. In contrast, applying a binary search to 4,000,000,000 elements performs a maximum of 32 comparisons. This is why Arrays contains binarySearch() methods and not also linearSearch() methods.
Following is a sampling of the Collections class’s collection-oriented class methods:

- static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c) returns the minimum element of collection c according to the natural ordering of its elements. For example, System.out.println(Collections.min(Arrays.asList(10, 3, 18, 25))); outputs 3. All of c’s elements must implement the Comparable interface. Furthermore, all elements must be mutually comparable. This method throws NoSuchElementException when c is empty.

- static void reverse(List<?> l) reverses the order of list l’s elements. For example, List<String> birds = Arrays.asList("Robin", "Oriole", "Bluejay"); Collections.reverse(birds); System.out.println(birds); results in [Bluejay, Oriole, Robin] as the output.

- static <T> List<T> singletonList(T o) returns an immutable list containing only object o. For example, list.removeAll(Collections.singletonList(null)); removes all null elements from list.

- static <T> Set<T> synchronizedSet(Set<T> s) returns a synchronized (thread-safe) set backed by the specified set s, for example, Set<String> ss = Collections.synchronizedSet(new HashSet<String>()); To guarantee serial access, it’s critical that all access to the backing set (s) is accomplished through the returned set.

- static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m) returns an unmodifiable view of map m, for example, Map<String, Integer> msi = Collections.unmodifiableMap(new HashMap<String, Integer>()); Query operations on the returned map “read through” to the specified map; and attempts to modify the returned map, whether direct or via its collection views, result in an UnsupportedOperationException.

Note For performance reasons, collections implementations are unsynchronized—unsynchronized collections have better performance than synchronized collections. To use a collection in a multithreaded context, however, you need to obtain a synchronized version of that collection. You obtain that version by calling a synchronized-prefixes method such as synchronizedSet().

You might be wondering about the purpose for the various “empty” class methods in the Collections class. For example, static final <T> List<T> emptyList() returns an immutable empty list, as in List<String> ls = Collections.emptyList(); These methods are present because they offer a useful alternative to returning null (and avoiding potential NullPointerExceptions) in certain contexts. Consider Listing 9-28.
Listing 9-28. Empty and Nonempty Lists of Birds

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

class Birds {
    private List<String> birds;

    Birds() {
        birds = Collections.emptyList();
    }

    Birds(String... birdNames) {
        birds = new ArrayList<String>();
        for (String birdName : birdNames)
            birds.add(birdName);
    }

    @Override
    public String toString() {
        return birds.toString();
    }
}

class EmptyListDemo {
    public static void main(String[] args) {
        Birds birds = new Birds();
        System.out.println(birds);
        birds = new Birds("Swallow", "Robin", "Bluejay", "Oriole");
        System.out.println(birds);
    }
}

Listing 9-28 declares a Birds class that stores the names of various birds in a list. This class provides two constructors: a no-argument constructor and a constructor that takes a variable number of String arguments identifying various birds.

The no-argument constructor invokes emptyList() to initialize its private birds field to an empty List of String—emptyList() is a generic method and the compiler infers its return type from its context.

If you're wondering about the need for emptyList(), look at the toString() method. Notice that this method evaluates birds.toString(). If you didn’t assign a reference to an empty List<String> to birds, birds would contain the null reference (the default value for this instance field when the object is created), and a NullPointerException instance would be thrown when attempting to evaluate birds.toString().
When you run this application (java EmptyListDemo), it generates the following output:

[]
[Swallow, Robin, Bluejay, Oriole]

The emptyList() method is implemented as follows: return (List<T>) EMPTY_LIST;. This statement returns the single List instance assigned to the EMPTY_LIST class field in the Collections class.

You might want to work with EMPTY_LIST directly, but you’ll run into an unchecked warning message if you do, because EMPTY_LIST is declared to be of the raw type List, and mixing raw and generic types leads to such messages. Although you could suppress the warning, you’re better off using the emptyList() method.

Suppose you add a void setBirds(List<String> birds) method to Birds and pass an empty list to this method, as in birds.setBirds(Collections.emptyList());. The compiler will respond with an error message stating that it requires the argument to be of type List<String>, but instead the argument is of type List<Object>. It does so because the compiler cannot figure out the proper type from this context, and so it chooses List<Object>.

This problem can be solved by explicitly specifying the type argument. Specify birds.setBirds(Collections.<String>emptyList());, where the formal type parameter list and its type argument appear after the member access operator and before the method name. The compiler will now know that the proper type argument is String and that emptyList() is to return List<String>.

Exploring the Legacy Collection APIs

Java 1.2 introduced the Collections Framework. Before the framework’s inclusion in Java, developers had two choices where collections were concerned: create their own frameworks, or use the Vector, Enumeration, Stack, Dictionary, Hashtable, Properties, and BitSet types, which were introduced by Java 1.0.

Vector is a concrete class that describes a growable array, much like ArrayList. Unlike an ArrayList instance, a Vector instance is synchronized. Vector has been generified and also retrofitted to support the Collections Framework, which makes statements such as List<String> list = new Vector<String>(); legal. You may need to perform similar assignments when working with legacy code that depends on Vector.

The Collections Framework provides Iterator for iterating over a collection’s elements. In contrast, Vector’s elements() method returns an instance of a class that implements the Enumeration interface for enumerating (iterating over and returning) a Vector instance’s elements via Enumeration’s hasMoreElements() and nextElement() methods. Consider the following example:

```java
Enumeration e = vector.elements();
while (e.hasMoreElements())
    System.out.println(e.nextElement());
```

Vector is subclassed by the concrete Stack class, which represents a LIFO data structure. Stack provides an E push(E item) method for pushing an object onto the stack, an E pop() method for popping an item off the top of the stack, and a few other methods, such as boolean empty() for determining whether or not the stack is empty.
Stack is a good example of bad API design. By inheriting from Vector, it’s possible to call Vector’s `void add(int index, E element)` method to add an element anywhere you wish and violate a Stack instance’s integrity. In hindsight, Stack should have used composition in its design: use a Vector instance to store a Stack instance’s elements.

Dictionary is an abstract superclass for subclasses that map keys to values. The concrete Hashtable class is Dictionary’s only subclass. As with Vector, HashTable instances are synchronized, HashTable has been generified, and HashTable has been retrofitted to support the Collections Framework.

Hashtable is subclassed by Properties, a concrete class representing a persistent set of properties (String-based key/value pairs that identify application settings). Properties provides `Object setProperty(String key, String value)` for storing a property and `String getProperty(String key)` for returning a property’s value.

Note Applications use properties for various purposes. For example, if your application has a graphical user interface, you could store the screen location and size of its main window in a file via a Properties object so that the application can restore the window’s location and size when it next runs.

Properties is another good example of bad API design. By inheriting from Hashtable, you can call Hashtable’s `V put(K key, V value)` method to store an entry with a non-String key and/or a non-String value. In hindsight, Properties should have leveraged composition: store a Properties instance’s elements in a Hashtable instance.

Note Chapter 4 discussed wrapper classes, which is how Stack and Properties should have been implemented.

Finally, BitSet is a concrete class that describes a variable-length set of bits. This class’s ability to represent bitsets of arbitrary length contrasts with the previously described integer-based, fixed-length bitset that is limited to a maximum number of members: 32 members for an int-based bitset, or 64 members for a long-based bitset.

BitSet provides a pair of constructors for initializing a BitSet instance: `BitSet()` initializes the instance to initially store an implementation-dependent number of bits, whereas `BitSet(int nbits)` initializes the instance to initially store `nbits` bits. BitSet also provides various methods, including the following:

- `void and(BitSet bs)` bitwise ANDs this bitset with `bs`. This bitset is modified such that a bit is set to 1 when it and the bit at the same position in `bs` are 1.
- `void andNot(BitSet bs)` sets all of the bits in this bitset to 0 whose corresponding bits are set to 1 in `bs`.
- `void clear()` sets all of the bits in this bitset to 0.
- Object clone() clones this bitset to produce a new bitset. The clone has exactly the same bits set to one as this bitset.

- boolean get(int bitIndex) returns the value of this bitset's bit as a Boolean true/false value (true for 1, false for 0) at the zero-based bitIndex. This method throws IndexOutOfBoundsException when bitIndex is less than 0.

- int length() returns the "logical size" of this bitset, which is the index of the highest 1 bit plus 1, or 0 if this bitset contains no 1 bits.

- void or(BitSet bs) bitwise inclusive ORs this bitset with bs. This bitset is modified such that a bit is set to 1 when it or the bit at the same position in bs is 1 or when both bits are 1.

- void set(int bitIndex, boolean value) sets the bit at the zero-based bitIndex to value (true is converted to 1; false is converted to 0). This method throws IndexOutOfBoundsException when bitIndex is less than 0.

- int size() returns the number of bits that are being used by this bitset to represent bit values.

- String toString() returns a string representation of this bitset in terms of the positions of bits that are 1, for example, {4, 5, 9, 10}.

- void xor(BitSet set) bitwise exclusive ORs this bitset with bs. This bitset is modified such that a bit is set to 1 when either it or the bit at the same position in bs (but not both) is 1.

Listing 9-29 presents an application that demonstrates some of these methods and gives you more insight into how the bitwise AND (&), bitwise inclusive OR (|), and bitwise exclusive OR (^) operators work.

Listing 9-29. Working with Variable-Length Bitsets

```java
import java.util.BitSet;

public class BitSetDemo {
    public static void main(String[] args) {
        BitSet bs1 = new BitSet();
        bs1.set(4, true);
        bs1.set(5, true);
        bs1.set(9, true);
        bs1.set(10, true);
        BitSet bsTemp = (BitSet) bs1.clone();
        dumpBitset("        ", bs1);
        BitSet bs2 = new BitSet();
        bs2.set(4, true);
        bs2.set(6, true);
        bs2.set(7, true);
        bs2.set(9, true);
        dumpBitset("        ", bs2);
        bs1.and(bs2);
```
dumpSeparator(Math.min(bs1.size(), 16));
dumpBitset("AND (&) ", bs1);
System.out.println();
bs1 = bsTemp;
dumpBitset(" ", bs1);
dumpBitset(" ", bs2);
bsTemp = (BitSet) bs1.clone();
bs1.or(bs2);
dumpSeparator(Math.min(bs1.size(), 16));
dumpBitset("OR (|) ", bs1);
System.out.println();
bs1 = bsTemp;
dumpBitset(" ", bs1);
dumpBitset(" ", bs2);
bsTemp = (BitSet) bs1.clone();
bs1.xor(bs2);
dumpSeparator(Math.min(bs1.size(), 16));
dumpBitset("XOR (^) ", bs1);
}

static void dumpBitset(String preamble, BitSet bs)
{
  System.out.print(preamble);
  int size = Math.min(bs.size(), 16);
  for (int i = 0; i < size; i++)
    System.out.print(bs.get(i) ? "1" : "0");
  System.out.print(" size(" + bs.size() + ")", length(" + bs.length() + ")");
  System.out.println();
}

static void dumpSeparator(int len)
{
  System.out.print(" ");
  for (int i = 0; i < len; i++)
    System.out.print("-");
  System.out.println();
}

Why did I specify Math.min(bs.size(), 16) in dumpBitset() and pass a similar expression to dumpSeparator()? I wanted to display exactly 16 bits and 16 dashes (for aesthetics), and needed to account for a bitset's size being less than 16. Although this doesn't happen with Oracle's and Google's BitSet classes, it might happen with some other variant.

When you run this application, it generates the following output:

```
    0000110001100000  size(64), length(11)
    0000101101000000  size(64), length(10)
----------------
AND (&) 0000100001000000  size(64), length(10)

    0000110001100000  size(64), length(11)
```

www.it-ebooks.info
**Caution**  Unlike Vector and Hashtable, BitSet is not synchronized. You must externally synchronize access to this class when using BitSet in a multithreaded context.

The Collections Framework has made Vector, Enumeration, Stack, Dictionary, and Hashtable obsolete. These types continue to be part of the standard class library to support legacy code. Also, the Preferences API has made Properties largely obsolete. Because BitSet is still relevant, this class continues to be improved (as recently as Java 7).

**Note**  It's not surprising that BitSet is being improved when you realize the usefulness of variable-length bitsets. Because of their compactness and other advantages, variable-length bitsets are often used to implement an operating system’s priority queues and facilitate memory page allocation. Unix-oriented file systems also use bitsets to facilitate the allocation of inodes (information nodes) and disk sectors. And bitsets are useful in Huffman coding, a data-compression algorithm for achieving lossless data compression.

**EXERCISES**

The following exercises are designed to test your understanding of Chapter 9's content.

1. What is a collection?
2. What is the Collections Framework?
3. The Collections Framework largely consists of what components?
4. Define comparable.
5. When would you have a class implement the Comparable interface?
6. What is a comparator and what is its purpose?
7. True or false: A collection uses a comparator to define the natural ordering of its elements.
8. What does the Iterable interface describe?
9. What does the Collection interface represent?
10. Identify a situation where Collection’s add() method would throw an instance of the 
   UnsupportedOperationException class.
11. Iterable’s iterator() method returns an instance of a class that implements the Iterator 
   interface. What methods does this interface provide?
12. What is the purpose of the enhanced for loop statement?
13. How is the enhanced for loop statement expressed?
14. True or false: The enhanced for loop works with arrays.
15. Define autoboxing.
17. What is a list?
18. What does a ListIterator instance use to navigate through a list?
19. What is a view?
20. Why would you use the subList() method?
21. What does the ArrayList class provide?
22. What does the LinkedList class provide?
23. Define node.
24. True or false: ArrayList provides faster element insertions and deletions than 
   LinkedList.
25. What is a set?
26. What does the TreeSet class provide?
27. What does the HashSet class provide?
28. True or false: To avoid duplicate elements in a hashset, your own classes must correctly override 
   equals() and hashCode().
29. What is the difference between HashSet and LinkedHashSet?
30. What does the EnumSet class provide?
31. Define sorted set.
32. What is a navigable set?
33. True or false: HashSet is an example of a sorted set.
34. Why would a sorted set’s add() method throw ClassCastException when you attempt to add an 
   element to the sorted set?
35. What is a queue?
36. True or false: Queue’s element() method throws NoSuchElementException when it is called on 
   an empty queue.
37. What does the PriorityQueue class provide?
38. What is a map?
39. What does the TreeMap class provide?
40. What does the HashMap class provide?
41. What does a hashtable use to map keys to integer values?
42. Continuing from the previous question, what are the resulting integer values called and what do they accomplish?
43. What is a hashtable’s capacity?
44. What is a hashtable’s load factor?
45. What is the difference between HashMap and LinkedHashMap?
46. What does the IdentityHashMap class provide?
47. What does the EnumMap class provide?
48. Define sorted map.
49. What is a navigable map?
50. True or false: TreeMap is an example of a sorted map.
51. What is the purpose of the Arrays class’s static <T> List<T> asList(T... array) method?
52. True or false: Binary search is slower than linear search.
53. Which Collections method would you use to return a synchronized variation of a hashset?
54. Identify the seven legacy collections-oriented types.
55. As an example of array list usefulness, create a JavaQuiz application that presents a multiple-choice-based quiz on Java features. The JavaQuiz class’s main() method first populates the array list with the entries in a QuizEntry array (such as new QuizEntry("What was Java’s original name?", new String[] { "Oak", "Duke", "J", "None of the above" }, 'A')). Each entry consists of a question, four possible answers, and the letter (A, B, C, or D) of the correct answer. main() then uses the array list’s iterator() method to return an Iterator instance and this instance’s hasNext() and next() methods to iterate over the list. Each of the iterations outputs the question and four possible answers and then prompts the user to enter the correct choice. After the user enters A, B, C, or D (via System.in.read()), main() outputs a message stating whether or not the user made the correct choice.
56. Why is (int) (f ^ (f >>> 32)) used instead of (int) (f ^ (f >> 32)) in the hash code generation algorithm?
57. Collections provides the static int frequency(Collection<? super T> c, Object o) method to return the number of collection c elements that are equal to o. Create a FrequencyDemo application that reads its command-line arguments and stores all arguments except for the last argument in a list and then calls frequency() with the list and last command-line argument as this method’s arguments. It then outputs this method’s return value (the number of occurrences of the last command-line argument in the previous command-line arguments). For example, java FrequencyDemo should output Number of occurrences of null = 0, and java FrequencyDemo how much wood could a woodchuck chuck if a woodchuck could chuck wood wood should output Number of occurrences of wood = 2.
Summary

A collection is a group of objects that are stored in an instance of a class designed for this purpose. To save you from having to create your own collections classes, Java provides the Collections Framework for representing and manipulating collections.

The Collections Framework largely consists of core interfaces, implementation classes, and the Arrays and Collections utility classes. The core interfaces make it possible to manipulate collections independently of their implementations.

Core interfaces include Iterable, Collection, List, Set, SortedSet, NavigableSet, Queue, Deque, Map, SortedMap, and NavigableMap. Collection extends Iterable; List, Set, and Queue each extend Collection; SortedSet extends Set; NavigableSet extends SortedSet; Deque extends Queue; SortedMap extends Map; and NavigableMap extends SortedMap.

Implementation classes include ArrayList, LinkedList, TreeSet, HashSet, LinkedHashSet, EnumSet, PriorityQueue, ArrayDeque, TreeMap, HashMap, LinkedHashMap, IdentityHashMap, WeakHashMap, and EnumMap. The name of each concrete class ends in a core interface name, identifying the core interface on which it is based.

The Collections Framework would not be complete without its Arrays and Collections utility classes. Each class supplies various class methods that implement useful algorithms in the contexts of arrays and collections. For example, Arrays lets you efficiently search and sort arrays, and Collections lets you obtain synchronized and unmodifiable collections.

Before Java 1.2’s introduction of the Collections Framework, developers could create their own frameworks or use the Vector, Enumeration, Stack, Dictionary, Hashtable, Properties, and BitSet types, which were introduced by Java 1.0.

The Collections Framework has made Vector, Enumeration, Stack, Dictionary, and Hashtable obsolete. Also, the Preferences API has made Properties largely obsolete. Because BitSet is still relevant, this class continues to be improved.

In Chapter 10 I explore the Concurrency Utilities API suite, which is Java’s concurrency counterpart to the Collections Framework.
Chapter 7 introduced Java's Threads API. Significant problems with Threads resulted in the development of the more powerful Concurrency Utilities Framework. In this chapter, I take you on a tour of this framework from Android’s perspective. Note that you won’t find a discussion of newer features, such as the Fork/Join Framework, because Android doesn’t support them.

Introducing the Concurrency Utilities

The low-level Threads API lets you create multithreaded applications that offer better performance and responsiveness over their single-threaded counterparts. However, there are problems:

- Low-level concurrency primitives, such as synchronized and wait()/notify(), are often hard to use correctly. Incorrect use of these primitives can result in race conditions, thread starvation, deadlock, and other hazards, which can be hard to detect and debug.

- Too much reliance on the synchronized primitive can lead to performance issues, which affect an application’s scalability. This is a significant problem for highly threaded applications such as web servers.

- Developers often need to use higher-level constructs such as thread pools and semaphores. Because these constructs aren’t included with Java’s low-level threading capabilities, developers have been forced to build their own constructs, which is a time-consuming and error-prone activity.
To address these problems, Java 5 introduced the Concurrency Utilities, a powerful and extensible framework of high-performance threading utilities such as thread pools and blocking queues. This framework consists of the following packages:

- `java.util.concurrent` contains utility types that are often used in concurrent programming, for example, executors, thread pools, and concurrent hashmaps.
- `java.util.concurrent.atomic` contains utility classes that support lock-free, thread-safe programming on single variables.
- `java.util.concurrent.locks` contains utility types for locking and waiting on conditions (objects that let threads suspend execution [wait] until notified by other threads that some Boolean state may now be true). Locking and waiting via these types provides better performance and is more flexible than doing so via Java’s monitor-based synchronization and wait/notification mechanisms.

This framework also introduces a `long nanoTime()` method to the `java.lang.System` class, which lets you access a nanosecond-granularity time source for making relative time measurements.

---

**Note** Two terms that are commonly encountered when exploring the Concurrency Utilities framework are parallelism and concurrency. According to Oracle’s “Multithreading Guide” (http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html), **parallelism** is “a condition that arises when at least two threads are **executing** simultaneously.” In contrast, **concurrency** is “a condition that exists when at least two threads are **making progress** [simultaneously or not. It is a] more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.”

---

The concurrency utilities can be classified as executors, synchronizers, concurrent collections, a locking framework, and atomic variables. I explore each category in the following sections.

### Exploring Executors

The Threads API lets you execute runnable tasks via expressions such as

```java
new java.lang.Thread(new RunnableTask()).start();
```

These expressions tightly couple task submission with the task’s execution mechanics (run on the current thread, a new thread, or a thread arbitrarily chosen from a pool [group] of threads).

**Note** A task is an object whose class implements the `java.lang.Runnable` interface (a runnable task) or the `java.util.concurrent.Callable` interface (a callable task).

---

www.it-ebooks.info
The concurrency-oriented utilities provide executors as a high-level alternative to low-level Threads API expressions for executing runnable tasks. An executor is an object whose class directly or indirectly implements the `java.util.concurrent.Executor` interface, which decouples task submission from task-execution mechanics.

*Note* The executor framework’s use of interfaces to decouple task submission from task-execution mechanics is analogous to the Collections Framework’s use of core interfaces to decouple lists, sets, queues, and maps from their implementations. Decoupling results in flexible code that’s easier to maintain.

Executor declares a solitary `void execute(Runnable runnable)` method that executes the runnable task named `runnable` at some point in the future. `execute()` throws `java.lang.NullPointerException` when `runnable` is null and `java.util.concurrent.RejectedExecutionException` when it cannot execute `runnable`.

*Note* `RejectedExecutionException` can be thrown when an executor is shutting down and doesn’t want to accept new tasks. Also, this exception can be thrown when the executor doesn’t have enough room to store the task (perhaps the executor uses a bounded blocking queue to store tasks and the queue is full. I discuss blocking queues later in this chapter).

The following example presents the `Executor` equivalent of the aforementioned `new Thread(new RunnableTask()).start();` expression:

```java
Executor executor = ...; // ... represents some executor creation
executor.execute(new RunnableTask());
```

Although `Executor` is easy to use, this interface is limited in various ways:

- Executor focuses exclusively on `Runnable`. Because `Runnable`’s `run()` method doesn’t return a value, there’s no convenient way for a runnable task to return a value to its caller.
- Executor doesn’t provide a way to track the progress of runnable tasks that are executing, cancel an executing runnable task, or determine when the runnable task finishes execution.
- Executor cannot execute a collection of runnable tasks.
- Executor doesn’t provide a way for an application to shut down an executor (much less to shut down an executor properly).

These limitations are addressed by the `java.util.concurrent.ExecutorService` interface, which extends `Executor` and whose implementation is typically a thread pool. Table 10-1 describes `ExecutorService`’s methods.
### Table 10-1. ExecutorService Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean awaitTermination(long timeout, TimeUnit unit)</td>
<td>Blocks (waits) until all tasks have finished after a shutdown request, the timeout (measured in unit time units) expires, or the current thread is interrupted, whichever happens first. Returns true when this executor has terminated and false when the timeout elapses before termination. This method throws java.lang.InterruptedException when interrupted.</td>
</tr>
<tr>
<td>&lt;T&gt; List&lt;Future&lt;T&gt;&gt; invokeAll(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks)</td>
<td>Executes each callable task in the tasks collection and returns a java.util.List of java.util.concurrent.Future instances that hold task statuses and results when all tasks complete—a task completes through normal termination or by throwing an exception. The List of Futures is in the same sequential order as the sequence of tasks returned by tasks’ iterator. This method throws InterruptedException when it’s interrupted while waiting, in which case unfinished tasks are canceled; NullPointerException when tasks or any of its elements is null; and RejectedExecutionException when any one of tasks’ tasks cannot be scheduled for execution.</td>
</tr>
<tr>
<td>&lt;T&gt; List&lt;Future&lt;T&gt;&gt; invokeAll(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks, long timeout, TimeUnit unit)</td>
<td>Executes each callable task in the tasks collection and returns a List of Future instances that hold task statuses and results when all tasks complete—a task completes through normal termination or by throwing an exception—or the timeout (measured in unit time units) expires. Tasks that are not completed at expiry are canceled. The List of Futures is in the same sequential order as the sequence of tasks returned by tasks’ iterator. This method throws InterruptedException when it’s interrupted while waiting, in which case unfinished tasks are canceled. It also throws NullPointerException when tasks, any of its elements, or unit is null; and throws RejectedExecutionException when any one of tasks’ tasks cannot be scheduled for execution.</td>
</tr>
<tr>
<td>&lt;T&gt; T invokeAny(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks)</td>
<td>Executes the given tasks, returning the result of an arbitrary task that’s completed successfully (in other words, without throwing an exception), if any does. On normal or exceptional return, tasks that haven’t completed are canceled. This method throws InterruptedException when it’s interrupted while waiting; NullPointerException when tasks or any of its elements is null; java.lang.IllegalArgumentException when tasks is empty; java.util.concurrent.ExecutionException when no task completes successfully; and RejectedExecutionException when none of the tasks can be scheduled for execution.</td>
</tr>
<tr>
<td>&lt;T&gt; T invokeAny(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks, long timeout, TimeUnit unit)</td>
<td>Executes the given tasks, returning the result of an arbitrary task that’s completed successfully (in other words, without throwing an exception), if any does before the timeout (measured in unit time units) expires—tasks that are not completed at expiry are canceled. On normal or exceptional return, tasks that have not completed are canceled. This method throws InterruptedException when it’s interrupted while waiting; NullPointerException when tasks, any of its elements, or unit is null; IllegalArgumentException when tasks is empty; java.util.concurrent.TimeoutException when the timeout elapses before any task successfully completes; ExecutionException when no task completes successfully; and RejectedExecutionException when none of the tasks can be scheduled for execution.</td>
</tr>
</tbody>
</table>

(continued)
Table 10-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isShutdown()</td>
<td>Returns true when this executor has been shut down; otherwise, returns false.</td>
</tr>
<tr>
<td>boolean isTerminated()</td>
<td>Returns true when all tasks have completed following shutdown; otherwise, returns false. This method will never return true prior to shutdown() or shutdownNow() being called.</td>
</tr>
<tr>
<td>void shutdown()</td>
<td>Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Calling this method has no effect after the executor has shut down. This method doesn’t wait for previously submitted tasks to complete execution. Use awaitTermination() when waiting is necessary.</td>
</tr>
<tr>
<td>List&lt;Runnable&gt; shutdownNow()</td>
<td>Attempts to stop all actively executing tasks, halt the processing of waiting tasks, and return a list of the tasks that were awaiting execution. There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via Thread.interrupt(), so any task that fails to respond to interrupts may never terminate.</td>
</tr>
<tr>
<td>&lt;T&gt; Future&lt;T&gt; submit(Callable&lt;T&gt; task)</td>
<td>Submits a callable task for execution and returns a Future instance representing task's pending results. The Future instance's get() method returns task's result on successful completion. This method throws RejectedExecutionException when task cannot be scheduled for execution and NullPointerException when task is null. If you would like to block immediately while waiting for a task to complete, you can use constructions of the form result = exec.submit(aCallable).get();</td>
</tr>
<tr>
<td>Future&lt;?&gt; submit(Runnable task)</td>
<td>Submits a runnable task for execution and returns a Future instance representing task's pending results. The Future instance's get() method returns task's result on successful completion. This method throws RejectedExecutionException when task cannot be scheduled for execution and NullPointerException when task is null.</td>
</tr>
<tr>
<td>&lt;T&gt; Future&lt;T&gt; submit(Runnable task, T result)</td>
<td>Submits a runnable task for execution and returns a Future instance whose get() method returns result on successful completion. This method throws RejectedExecutionException when task cannot be scheduled for execution and NullPointerException when task is null.</td>
</tr>
</tbody>
</table>

Table 10-1 refers to java.util.concurrent.TimeUnit, an enum that represents time durations at given units of granularity: DAYS, HOURS, MICROSECONDS, MILLISECONDS, MINUTES, NANOSECONDS, and SECONDS. Furthermore, TimeUnit declares methods for converting across units (such as long toHours(long duration)), and for performing timing and delay operations (such as void sleep(long timeout)) in these units.

Table 10-1 also refers to callable tasks, which are analogous to runnable tasks. Unlike Runnable, whose void run() method cannot throw checked exceptions, Callable&lt;V&gt; declares a V call() method that returns a value and which can throw checked exceptions because call() is declared with a throws Exception clause.
Finally, Table 10-1 refers to the Future interface, which represents the result of an asynchronous computation. The result is known as a future, because it typically will not be available until some moment in the future. Future, whose generic type is Future<V>, provides methods for canceling a task, for returning a task’s value, and for determining whether or not the task has finished. Table 10-2 describes Future’s methods.

**Table 10-2. Future Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean cancel(boolean mayInterruptIfRunning)</td>
<td>Attempts to cancel execution of this task and returns true when the task was canceled; otherwise, returns false (perhaps the task completed normally before this method was called). The cancellation attempt fails when the task is done, canceled, or couldn’t be canceled for another reason. If successful and this task hadn’t started when cancel() was called, the task should never run. If the task has started, mayInterruptIfRunning determines whether (true) or not (false) the thread executing this task should be interrupted in an attempt to stop the task. After returning, subsequent calls to isDone() always return true; isCancelled() always return true when cancel() returns true.</td>
</tr>
<tr>
<td>V get()</td>
<td>Waits if necessary for the task to complete and then returns the result. This method throws java.util.concurrent.CancellationException when the task was canceled prior to this method being called, ExecutionException when the task threw an exception, and InterruptedException when the current thread was interrupted while waiting.</td>
</tr>
<tr>
<td>V get(long timeout, TimeUnit unit)</td>
<td>Waits at most timeout units (as specified by unit) for the task to complete and then returns the result (if available). This method throws CancellationException when the task was canceled prior to this method being called, ExecutionException when the task threw an exception, InterruptedException when the current thread was interrupted while waiting, and TimeoutException when this method’s timeout value expires (the wait times out).</td>
</tr>
<tr>
<td>boolean isCancelled()</td>
<td>Returns true when this task was canceled before it completed normally; otherwise, returns false.</td>
</tr>
<tr>
<td>boolean isDone()</td>
<td>Returns true when this task completed; otherwise, returns false. Completion may be due to normal termination, an exception, or cancellation; this method returns true in all of these cases.</td>
</tr>
</tbody>
</table>

Suppose that you intend to write an application whose graphical user interface lets the user enter a word. After the user enters the word, the application presents this word to several online dictionaries and obtains each dictionary’s entry. These entries are subsequently displayed to the user.
Because online access can be slow, and because the user interface should remain responsive (perhaps the user might want to end the application), you offload the “obtain word entries” task to an executor that runs this task on a separate thread. The following example uses ExecutorService, Callable, and Future to accomplish this objective:

```java
ExecutorService executor = ...;  // ... represents some executor creation
Future<String[]> taskFuture = executor.submit(new Callable<String[]>()
{
    @Override
    public String[] call()
    {
        String[] entries = ...;
        // Access online dictionaries
        // with search word and populate
        // entries with their resulting
        // entries.
        return entries;
    }
});

// Do stuff.
String entries = taskFuture.get();
```

After obtaining an executor in some manner (you will learn how to do this shortly), the example’s main thread submits a callable task to the executor. The `submit()` method immediately returns with a reference to a `Future` object for controlling task execution and accessing results. The main thread ultimately calls this object’s `get()` method to get these results.

**Note** The `java.util.concurrent.ScheduledExecutorService` interface extends `ExecutorService` and describes an executor that lets you schedule tasks to run once or to execute periodically after a given delay.

Although you could create your own `Executor`, `ExecutorService`, and `ScheduledExecutorService` implementations (such as class `DirectExecutor` implements `Executor` { `@Override` public void execute(Runnable r) { r.run(); } }—run executor directly on the calling thread), there’s a simpler alternative: `java.util.concurrent.Executors`.

**Tip** If you intend to create your own `ExecutorService` implementations, you will find it helpful to work with the `java.util.concurrent.AbstractExecutorService` and `java.util.concurrent.FutureTask` classes.
The Executors utility class declares several class methods that return instances of various ExecutorService and ScheduledExecutorService implementations (and other kinds of instances). This class's static methods accomplish the following tasks:

- Create and return an ExecutorService instance that's configured with commonly used configuration settings.
- Create and return a ScheduledExecutorService instance that's configured with commonly used configuration settings.
- Create and return a “wrapped” ExecutorService or ScheduledExecutorService instance that disables reconfiguration of the executor service by making implementation-specific methods inaccessible.
- Create and return a java.util.concurrent.ThreadFactory instance (that is, an instance of a class that implements the ThreadFactory interface) for creating new threads.
- Create and return a Callable instance out of other closure-like forms so that it can be used in execution methods that require Callable arguments (such as ExecutorService's submit(Callable) method). (Wikipedia's “Closure (computer science)” entry [http://en.wikipedia.org/wiki/Closure_(computer_science)] introduces the topic of closures.)

For example, static ExecutorService newFixedThreadPool(int nThreads) creates a thread pool that reuses a fixed number of threads operating off of a shared unbounded queue. At most, nThreads threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue for an available thread.

If any thread terminates because of a failure during execution before the executor shuts down, a new thread will take its place when needed to execute subsequent tasks. The threads in the pool will exist until the executor is explicitly shut down. This method throws IllegalArgumentException when you pass zero or a negative value to nThreads.

Note  Thread pools are used to eliminate the overhead from having to create a new thread for each submitted task. Thread creation isn’t cheap, and having to create many threads could severely impact an application’s performance.

You would commonly use executors, runnables, callables, and futures in file and network input/output contexts. Performing a lengthy calculation offers another scenario where you could use these types. For example, Listing 10-1 uses an executor, a callable, and a future in a calculation context of Euler's number e (2.71828...).

Listing 10-1. Calculating Euler’s Number

```java
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
```

www.it-ebooks.info
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CalculateE
{
    final static int LASTITER = 17;

    public static void main(String[] args)
    {
        ExecutorService executor = Executors.newFixedThreadPool(1);
        Callable<BigDecimal> callable;
        callable = new Callable<BigDecimal>()
        {
            @Override
            public BigDecimal call()
            {
                MathContext mc = new MathContext(100,
                        RoundingMode.HALF_UP);
                BigDecimal result = BigDecimal.ZERO;
                for (int i = 0; i <= LASTITER; i++)
                {
                    BigDecimal factorial = factorial(new BigDecimal(i));
                    BigDecimal res = BigDecimal.ONE.divide(factorial, mc);
                    result = result.add(res);
                }
                return result;
            }
        }
        Future<BigDecimal> taskFuture = executor.submit(callable);
        try
        {
            while (!taskFuture.isDone())
                System.out.println("waiting");
            System.out.println(taskFuture.get());
        }
        catch(ExecutionException ee)
        {
            System.err.println("task threw an exception");
            System.err.println(ee);
        }
    }

    public BigDecimal factorial(BigDecimal n)
    {
        if (n.equals(BigDecimal.ZERO))
            return BigDecimal.ONE;
        else
            return n.multiply(factorial(n.subtract(BigDecimal.ONE)));
    }
}
catch(InterruptedException ie)
{
    System.err.println("interrupted while waiting");
}
executor.shutdownNow();
}
}

The main thread that executes Listing 10-1's main() method first obtains an executor by calling Executors' newFixedThreadPool() method. It then instantiates an anonymous class that implements the Callable interface and submits this task to the executor, receiving a Future instance in response.

After submitting a task, a thread typically does some other work until it requires the task's result. I've chosen to simulate this work by having the main thread repeatedly output a waiting message until the Future instance's isDone() method returns true. (In a realistic application, I would avoid this looping.) At this point, the main thread calls the instance's get() method to obtain the result, which is then output. The main thread then shuts down the executor.

**Caution** It's important to shut down the executor after it completes; otherwise, the application might not end. The previous executor accomplishes this task by calling shutdownNow(). (You could also use the shutdown() method.)

The callable's call() method calculates \( e \) by evaluating the mathematical power series \( e = 1 / 0! + 1 / 1! + 1 / 2! + \ldots \). This series can be evaluated by summing \( 1 / n! \), where \( n \) ranges from 0 to infinity.

call() first instantiates java.math.MathContext to encapsulate a *precision* (number of digits) and a rounding mode. I chose 100 as an upper limit on \( e \)'s precision, and I also chose HALF_UP as the rounding mode.

**Tip** Increase the precision as well as LASTITER's value to converge the series to a lengthier and more accurate approximation of \( e \).

call() next initializes a java.math.BigDecimal local variable named result to BigDecimal.ZERO. It then enters a loop that calculates a factorial, divides BigDecimal.ONE by the factorial, and adds the division result to result.

The divide() method takes the MathContext instance as its second argument to ensure that the division doesn't result in a *nonterminating decimal expansion* (the quotient result of the division cannot be represented exactly—0.3333333... for example), which throws java.lang.ArithmeticException (to alert the caller to the fact that the quotient cannot be represented exactly), which the executor rethrows as ExecutionException.
When you run this application, you should observe output similar to the following:

```
waiting
waiting
waiting
waiting
2.7182818284590450705160477958486050611789796352510326989007350040652250425048433140558879743442457
41730039454062711
```

**Exploring Synchronizers**

The Threads API offers synchronization primitives for synchronizing thread access to critical sections. Because it can be difficult to write synchronized code correctly that’s based on these primitives, Concurrency Utilities includes *synchronizers*, classes that facilitate common forms of synchronization.

Four commonly used synchronizers are countdown latches, cyclic barriers, exchangers, and semaphores. I explore each synchronizer in this section.

**Countdown Latches**

A *countdown latch* causes one or more threads to wait at a “gate” until another thread opens this gate, at which point these other threads can continue. It consists of a count and operations for “causing a thread to wait until the count reaches zero” and “decrementing the count.”

The `java.util.concurrent.CountDownLatch` class implements the countdown latch synchronizer. You initialize a `CountDownLatch` instance to a specific count by invoking this class’s `CountDownLatch(int count)` constructor. This constructor throws `IllegalArgumentException` when the value passed to `count` is negative.

`CountDownLatch` also offers the following methods:

- `void await()`: Forces the calling thread to wait until the latch has counted down to zero, unless the thread is interrupted, in which case `InterruptedException` is thrown. This method returns immediately when the count is zero.

- `boolean await(long timeout, TimeUnit unit)`: Forces the calling thread to wait until the latch has counted down to zero or the specified `timeout` value in `unit` time-units has expired, or the thread is interrupted, in which case `InterruptedException` is thrown. This method returns immediately when the count is zero. It returns true when the count reaches zero or false when the waiting time elapses.

- `void countDown()`: Decrements the count, releasing all waiting threads when the count reaches zero. Nothing happens when the count is already zero when this method is called.
- long getCount(): Returns the current count. This method is useful for testing and debugging.

- String toString(): Returns a string identifying this latch as well as its state. The state, in brackets, includes string literal "Count =" followed by the current count.

You'll often use a countdown latch to ensure that threads start working at approximately the same time. For example, check out Listing 10-2.

Listing 10-2. Using a Countdown Latch to Trigger a Coordinated Start

```java
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CountDownLatchDemo {
    final static int NTHREADS = 3;

    public static void main(String[] args) {
        final CountDownLatch startSignal = new CountDownLatch(1);
        final CountDownLatch doneSignal = new CountDownLatch(NTHREADS);
        Runnable r = new Runnable() {
            @Override
            public void run() {
                try {
                    report("entered run()");
                    startSignal.await(); // wait until told to proceed
                    report("doing work");
                    Thread.sleep((int) (Math.random() * 1000));
                    doneSignal.countDown(); // reduce count on which
                    // main thread is waiting
                } catch (InterruptedException ie) {
                    System.err.println(ie);
                }
            }

            void report(String s) {
                System.out.println(System.currentTimeMillis() + ": " +
                                    Thread.currentThread() + ": " + s);
            }
        };
```
```
ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
for (int i = 0; i < NTHREADS; i++)
    executor.execute(r);
try {
    System.out.println("main thread doing something");
    Thread.sleep(1000); // sleep for 1 second
    startSignal.countDown(); // let all threads proceed
    System.out.println("main thread doing something else");
    doneSignal.await(); // wait for all threads to finish
    executor.shutdownNow();
} catch (InterruptedException ie) {
    System.err.println(ie);
}
```

Listing 10-2’s main thread first creates a pair of countdown latches. The `startSignal` countdown latch prevents any worker thread from proceeding until the main thread is ready for them to proceed. The `doneSignal` countdown latch causes the main thread to wait until all worker threads have finished.

The main thread next creates a runnable whose `run()` method is executed by subsequently-created worker threads.

The `run()` method first outputs an initial message and then calls `startSignal`’s `await()` method to wait for this countdown latch’s count to read zero before it can proceed. Once this happens, `run()` outputs a message that indicates work being done and sleeps for a random period of time (0 through 999 milliseconds) to simulate this work.

At this point, `run()` invokes `doneSignal`’s `countDown()` method to decrement this latch’s count. Once this count reaches zero, the main thread waiting on this signal will continue, shutting down the executor and terminating the application.

After creating the runnable, the main thread obtains an executor that’s based on a thread pool of `NTHREADS` threads, and then calls the executor’s `execute()` method `NTHREADS` times, passing the runnable to each of the `NTHREADS` pool-based threads. This action starts the worker threads, which enter `run()`.

Next, the main thread outputs a message and sleeps for one second to simulate doing additional work (giving all the worker threads a chance to have entered `run()` and invoke `startSignal.await()`), invokes `startSignal`’s `countDown()` method to cause the worker threads to start running, outputs a message to indicate that it’s doing something else, and invokes `doneSignal`’s `await()` method to wait for this countdown latch’s count to reach zero before it can proceed.

Compile Listing 10-2:

```
javac CountDownLatchDemo.java
```
When you run this application (java CountDownLatchDemo), you'll observe output similar to the following:

```java
main thread doing something
1384281251694: Thread[pool-1-thread-1,5,main]: entered run()
1384281251694: Thread[pool-1-thread-2,5,main]: entered run()
1384281251694: Thread[pool-1-thread-3,5,main]: entered run()
main thread doing something else
1384281252723: Thread[pool-1-thread-3,5,main]: doing work
1384281252723: Thread[pool-1-thread-2,5,main]: doing work
1384281252723: Thread[pool-1-thread-1,5,main]: doing work
```

## Cyclic Barriers

A cyclic barrier lets a set of threads wait for each other to reach a common barrier point. The barrier is cyclic because it can be reused after the waiting threads are released. This synchronizer is useful in applications involving a fixed-size party of threads that must occasionally wait for each other.

The java.util.concurrent.CyclicBarrier class implements the cyclic barrier synchronizer. You initialize a CyclicBarrier instance to a specific number of parties (threads working toward a common goal) by invoking this class's CyclicBarrier(int parties) constructor. This constructor throws IllegalArgumentException when the value passed to parties is less than 1.

Alternatively, you can invoke the CyclicBarrier(int parties, Runnable barrierAction) constructor to initialize a cyclic barrier to a specific number of parties and a barrierAction that's executed when the barrier is tripped. In other words, when parties - 1 threads are waiting and one more thread arrives, that thread executes barrierAction and then all threads proceed. This runnable is useful for updating shared state before any of the threads continue. This constructor throws IllegalArgumentException when the value passed to parties is less than 1. (The former constructor invokes this constructor passing null to barrierAction; no runnable will be executed when the barrier is tripped.)

CyclicBarrier also offers the following methods:

- **int await()**: Forces the calling thread to wait until all parties have invoked await() on this cyclic barrier. The calling thread will also stop waiting when it or another waiting thread is interrupted, another thread times out while waiting, or another thread invokes reset() on this cyclic barrier. If the calling thread has its interrupted status set on entry or is interrupted while waiting, this method throws InterruptedException. It throws java.util.concurrent.BrokenBarrierException when the barrier is reset() while any thread is waiting or when the barrier is broken when await() is invoked. When any thread is interrupted while waiting, all other waiting threads throw BrokenBarrierException and the barrier is placed into the broken state. If the calling thread is the last thread to arrive and a non-null barrierAction was supplied in the constructor, the calling thread executes this runnable before allowing the other threads to continue. This method returns the arrival index of the calling thread, where index getParties() - 1 indicates the first thread to arrive and zero indicates the last thread to arrive.
CHAPTER 10: Exploring the Concurrency Utilities

- int await(long timeout, TimeUnit unit): This method is equivalent to the previous method except that it lets you specify how long the calling thread is willing to wait. This method throws TimeoutException when this timeout expires while the thread is waiting.

- int getNumberWaiting(): Returns the number of parties that are currently waiting at the barrier. This method is useful for debugging and in partnership with assertions.

- int getParties(): Returns the number of parties that are required to trip the barrier.

- boolean isBroken(): Returns true when one or more parties broke out of this barrier because of interruption or timeout since the cyclic barrier was constructed or the last reset, or when a barrier action failed because of an exception; otherwise, returns false.

- void reset(): Resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException. Note that resets after a breakage has occurred for other reasons can be complicated to carry out; threads need to resynchronize in some other way, and choose one thread to perform the reset. Therefore, it might be preferable to create a new barrier for subsequent use.

Cyclic barriers are useful in parallel decomposition scenarios where a lengthy task is divided into subtasks whose individual results are later merged into the overall result of the task. CyclicBarrier’s Javadoc presents example code that’s completed in Listing 10-3.

Listing 10-3. Using a Cyclic Barrier to Decompose a Task Into Subtasks

```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {
    public static void main(String[] args) {
        float[][] matrix = new float[3][3];
        int counter = 0;
        for (int row = 0; row < matrix.length; row++)
            for (int col = 0; col < matrix[0].length; col++)
                matrix[row][col] = counter++;
        dump(matrix);
        System.out.println();
        Solver solver = new Solver(matrix);
        System.out.println();
        dump(matrix);
    }
```
static void dump(float[][] matrix) {
    for (int row = 0; row < matrix.length; row++) {
        for (int col = 0; col < matrix[0].length; col++) {
            System.out.print(matrix[row][col] + " ");
        }
        System.out.println();
    }
}

class Solver {
    final int N;
    final float[][] data;
    final CyclicBarrier barrier;

    class Worker implements Runnable {
        int myRow;
        boolean done = false;

        Worker(int row) {
            myRow = row;
        }

        boolean done() {
            return done;
        }

        void processRow(int myRow) {
            System.out.println("Processing row: " + myRow);
            for (int i = 0; i < N; i++) {
                data[myRow][i] *= 10;
                done = true;
            }
        }

        @Override
        public void run() {
            while (!done()) {
                processRow(myRow);
                try {
                    barrier.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
catch (InterruptedException ex)
{
    return;
}
catch (BrokenBarrierException ex)
{
    return;
}
}

public Solver(float[][][] matrix)
{
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N,
        new Runnable()
        {
            @Override
            public void run()
            {
                mergeRows();
            }
        });

    for (int i = 0; i < N; ++i)
        new Thread(new Worker(i)).start();

    waitUntilDone();
}

void mergeRows()
{
    System.out.println("merging");
    synchronized("abc")
    {
        "abc".notify();
    }
}

void waitUntilDone()
{
    synchronized("abc")
    {
        try
        {
            System.out.println("main thread waiting");
            "abc".wait();
            System.out.println("main thread notified");
        }
    }
}
Listing 10-3’s main thread first creates a square matrix of floating-point values and dumps this matrix to the standard output stream. This thread then instantiates the Solver class, which creates a separate thread for performing a calculation on each row. The modified matrix is then dumped.

Solver presents a constructor that receives its matrix argument and saves its reference in field data along with the number of rows in field N. The constructor then creates a cyclic barrier with N parties and a barrier action that’s responsible for merging all of the rows into a final matrix. Finally, the constructor creates a worker thread that executes a separate Worker runnable that’s responsible for processing a single row in the matrix. The constructor then waits until the workers are finished.

Worker’s run() method repeatedly invokes processRow() on its specific row until done() returns true, which (in this example) it does after processRow() executes one time. After processRow() returns, which indicates that the row has been processed, the worker thread invokes await() on the cyclic barrier; it cannot proceed.

At some point, all of the worker threads will have invoked await(). When the final thread, which processes the final row in the matrix, invokes await(), it will trigger the barrier action, which merges all processed rows into a final matrix. In this example, a merger isn’t required, but it would be required in more complex examples.

The final task performed by mergeRows() is to notify the main thread that invoked Solver’s constructor. This thread is waiting on the monitor associated with String object "abc". A call to notify() suffices to wake up the waiting thread, which is the only thread waiting on this monitor.

Compile Listing 10-3:

code
javac CyclicBarrierDemo.java

doctest
When you run this application (java CyclicBarrierDemo), you'll observe output similar to the following:

code
0.0 1.0 2.0
3.0 4.0 5.0
6.0 7.0 8.0

main thread waiting
Processing row: 1
Processing row: 2
Processing row: 0
merging
main thread notified

0.0 10.0 20.0
30.0 40.0 50.0
60.0 70.0 80.0
Exchangers

An exchanger provides a synchronization point where threads can swap objects. Each thread presents some object on entry to the exchanger’s exchange() method, matches with a partner thread, and receives its partner’s object on return. Exchangers can be useful in applications such as genetic algorithms (see http://en.wikipedia.org/wiki/Genetic_algorithm) and pipeline designs.

The generic java.util.concurrent.Exchanger<V> class implements the exchanger synchronizer. You initialize an exchanger by invoking the Exchanger() constructor. You then invoke either of the following methods to perform an exchange:

- V exchange(V x): Waits for another thread to arrive at this exchange point (unless the calling thread is interrupted), and then transfers the given object to it, receiving the other thread’s object in return. If another thread is already waiting at the exchange point, it's resumed for thread scheduling purposes and receives the object passed in by the calling thread. The current thread returns immediately, receiving the object passed to the exchanger by the other thread. This throws InterruptedException when the calling thread is interrupted.

- V exchange(V x, long timeout, TimeUnit unit): This method is equivalent to the previous method except that it lets you specify how long the calling thread is willing to wait. It throws TimeoutException when this timeout expires while the thread is waiting.

Listing 10-4 expands on the “repeated buffer filling and emptying” Exchanger example presented in Exchanger’s Javadoc.

Listing 10-4. Using an Exchanger to Swap Buffers

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Exchanger;

public class ExchangerDemo
{
    static Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();

    static DataBuffer initialEmptyBuffer = new DataBuffer();
    static DataBuffer initialFullBuffer = new DataBuffer("I");

    public static void main(String[] args)
    {
        class FillingLoop implements Runnable
        {
            int count = 0;
```
@Override
public void run()
{
    DataBuffer currentBuffer = initialEmptyBuffer;
    try
    {
        while (true)
        {
            addToBuffer(currentBuffer);
            if (currentBuffer.isFull())
            {
                System.out.println("filling thread wants to exchange");
                currentBuffer = exchanger.exchange(currentBuffer);
                System.out.println("filling thread receives an exchange");
            }
        }
    }
    catch (InterruptedException ie)
    {
        System.out.println("filling thread interrupted");
    }
}

void addToBuffer(DataBuffer buffer)
{
    String item = "NI" + count++;
    System.out.println("Adding: " + item);
    buffer.add(item);
}

class EmptyingLoop implements Runnable
{
    @Override
    public void run()
    {
        DataBuffer currentBuffer = initialFullBuffer;
        try
        {
            while (true)
            {
                takeFromBuffer(currentBuffer);
                if (currentBuffer.isEmpty())
                {
                    System.out.println("emptying thread wants to exchange");
                    currentBuffer = exchanger.exchange(currentBuffer);
                    System.out.println("emptying thread receives an exchange");
                }
            }
        }
    }
}
catch (InterruptedException ie)
{
    System.out.println("emptying thread interrupted");
}

void takeFromBuffer(DataBuffer buffer)
{
    System.out.println("taking: " + buffer.remove());
}
new Thread(new EmptyingLoop()).start();
new Thread(new FillingLoop()).start();

}
String remove() {
    if (!isEmpty())
        return items.remove(0);
    return null;
}

Listing 10-4’s main thread creates an exchanger and a pair of buffers via static field initializers. It then instantiates the EmptyingLoop and FillingLoop local classes and passes these runnables to new Thread instances whose threads are then started. Each runnable’s run() method enters an infinite loop that repeatedly adds to or removes from its buffer. When the buffer is full or empty, the exchanger is used to swap these buffers and the filling or emptying continues.

Compile Listing 10-4:

javac ExchangerDemo.java

When you run this application (java ExchangerDemo), you’ll observe a prefix of the output similar to the following:

<table>
<thead>
<tr>
<th>Adding</th>
<th>I0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding</td>
<td>I1</td>
</tr>
<tr>
<td>Adding</td>
<td>I2</td>
</tr>
<tr>
<td>Adding</td>
<td>I3</td>
</tr>
<tr>
<td>Adding</td>
<td>I4</td>
</tr>
<tr>
<td>Adding</td>
<td>I5</td>
</tr>
<tr>
<td>Adding</td>
<td>I6</td>
</tr>
<tr>
<td>Adding</td>
<td>I7</td>
</tr>
<tr>
<td>Adding</td>
<td>I8</td>
</tr>
<tr>
<td>Adding</td>
<td>I9</td>
</tr>
<tr>
<td>taking:</td>
<td>I0</td>
</tr>
<tr>
<td>taking:</td>
<td>I1</td>
</tr>
<tr>
<td>taking:</td>
<td>I2</td>
</tr>
<tr>
<td>taking:</td>
<td>I3</td>
</tr>
<tr>
<td>taking:</td>
<td>I4</td>
</tr>
<tr>
<td>taking:</td>
<td>I5</td>
</tr>
<tr>
<td>taking:</td>
<td>I6</td>
</tr>
<tr>
<td>taking:</td>
<td>I7</td>
</tr>
<tr>
<td>taking:</td>
<td>I8</td>
</tr>
<tr>
<td>taking:</td>
<td>I9</td>
</tr>
<tr>
<td>emptying thread wants to exchange</td>
<td></td>
</tr>
</tbody>
</table>
Semaphores

A semaphore maintains a set of permits for restricting the number of threads that can access a limited resource. A thread attempting to acquire a permit when no permits are available blocks until some other thread releases a permit.

**Note** Semaphores whose current values can be incremented past 1 are known as counting semaphores, whereas semaphores whose current values can be only 0 or 1 are known as binary semaphores or mutexes. In either case, the current value cannot be negative.

The java.util.concurrent.Semaphore class implements this synchronizer and conceptualizes a semaphore as an object maintaining a set of permits. You initialize a semaphore by invoking the Semaphore(int permits) constructor where permits specifies the number of available permits. The resulting semaphore’s fairness policy is set to false (unfair). Alternatively, you can invoke the Semaphore(int permits, boolean fair) constructor to also set the semaphore’s fairness setting to true (fair).
When the fairness setting is false, Semaphore makes no guarantees about the order in which threads acquire permits. In particular, barging is permitted; that is, a thread invoking acquire() can be allocated a permit ahead of a thread that has been waiting; logically the new thread places itself at the head of the queue of waiting threads. When fair is set to true, the semaphore guarantees that threads invoking any of the acquire() methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in, first-out; or FIFO). Because FIFO ordering necessarily applies to specific internal points of execution within these methods, it's possible for one thread to invoke acquire() before another thread, but reach the ordering point after the other thread and similarly upon return from the method. Also, the untimed tryAcquire() methods don't honor the fairness setting; they'll take any available permits.

Generally, semaphores used to control resource access should be initialized as fair to ensure that no thread is starved out from accessing a resource. When using semaphores for other kinds of synchronization control, the throughput advantages of unfair ordering often outweigh fairness considerations.

Semaphore also offers the following methods:

- **void acquire()**: Acquires a permit from this semaphore, blocking until one is available or the calling thread is interrupted. InterruptedException is thrown when interrupted.

- **void acquire(int permits)**: Acquires permits permits from this semaphore, blocking until they are available or the calling thread is interrupted. InterruptedException is thrown when interrupted; IllegalArgumentException is thrown when permits is less than zero.

- **void acquireUninterruptibly()**: Acquires a permit, blocking until one is available.

- **void acquireUninterruptibly(int permits)**: Acquires permits permits, blocking until they are all available. IllegalArgumentException is thrown when permits is less than zero.

- **int availablePermits()**: Returns the current number of available permits. This method is useful for debugging and testing.

- **int drainPermits()**: Acquires and returns a count of all permits that are immediately available.

- **int getQueueLength()**: Returns an estimate of the number of threads waiting to acquire permits. The returned value is only an estimate because the number of threads may change dynamically while this method traverses internal data structures. This method is designed for use in monitoring system state and not for synchronization control.

- **boolean hasQueuedThreads()**: Queries whether any threads are waiting to acquire permits. Because cancellations may occur at any time, a true return value doesn't guarantee that another thread will ever acquire permits. This method is designed primarily for use in monitoring system state. It returns true when there may be other waiting threads.
boolean isFair(): Returns the fairness setting (true for fair and false for unfair).

void release(): Releases a permit, returning it to the semaphore. The number of available permits is increased by one. If any threads are trying to acquire a permit, one thread is selected and given the permit that was just released. That thread is reenabled for thread scheduling purposes.

void release(int permits): Releases permits permits, returning them to the semaphore. The number of available permits is increased by permits. If any threads are trying to acquire permits, one is selected and given the permits that were just released. If the number of available permits satisfies that thread's request, the thread is reenabled for thread scheduling purposes; otherwise, the thread will wait until sufficient permits are available. If there are permits available after this thread's request has been satisfied, those permits are assigned to other threads trying to acquire permits. IllegalArgumentException is thrown when permits is less than zero.

String toString(): Returns a string identifying this semaphore as well as its state. The state, in brackets, includes the string literal "Permits =" followed by the number of permits.

boolean tryAcquire(): Acquires a permit from this semaphore but only when one is available at the time of invocation. Returns true when the permit was acquired. Otherwise, returns immediately with value false.

boolean tryAcquire(int permits): Acquires permits permits from this semaphore, but only when they are available at the time of invocation. Returns true when the permits were acquired. Otherwise, returns immediately with value false. IllegalArgumentException is thrown when permits is less than zero.

boolean tryAcquire(int permits, long timeout, TimeUnit unit): Like the previous method, but the calling thread waits when permits permits aren't available. The wait ends when the permits become available, the timeout expires, or the calling thread is interrupted, in which case InterruptedException is thrown.

boolean tryAcquire(long timeOut, TimeUnit unit): Like tryAcquire(int permits), but the calling thread waits until a permit is available. The wait ends when the permit becomes available, the timeout expires, or the calling thread is interrupted, in which case InterruptedException is thrown.

Listing 10-5 expands on the “controlling access to a pool of items” Semaphore example presented in Semaphore’s Javadoc.

Listing 10-5. Using a Counting Semaphore to Control Access to a Pool of Items

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Semaphore;
public class SemaphoreDemo
{
    public static void main(String[] args)
    {
        final Pool pool = new Pool();
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                try
                {
                    while (true)
                    {
                        String item;
                        System.out.println(name + " acquiring " +
                                             (item = pool.getItem()));
                        Thread.sleep(200 + (int) (Math.random() * 100));
                        System.out.println(name + " putting back " + item);
                        pool.putItem(item);
                    }
                }
                catch (InterruptedException ie)
                {
                    System.out.println(name + "interrupted");
                }
            }
        };
        ExecutorService[] executors = new ExecutorService[Pool.MAX_AVAILABLE+1];
        for (int i = 0; i < executors.length; i++)
        {
            executors[i] = Executors.newSingleThreadExecutor();
            executors[i].execute(r);
        }
    }
}

final class Pool
{
    public static final int MAX_AVAILABLE = 10;

    private Semaphore available = new Semaphore (MAX_AVAILABLE, true);

    private String[] items;

    private boolean[] used = new boolean[MAX_AVAILABLE];
Pool()
{
    items = new String[MAX_AVAILABLE];
    for (int i = 0; i < items.length; i++)
        items[i] = "I" + i;
}

String getItem() throws InterruptedException
{
    available.acquire();
    return getNextAvailableItem();
}

void putItem(String item)
{
    if (markAsUnused(item))
        available.release();
}

private synchronized String getNextAvailableItem()
{
    for (int i = 0; i < MAX_AVAILABLE; ++i)
    {
        if (!used[i])
        {
            used[i] = true;
            return items[i];
        }
    }
    return null; // not reached
}

private synchronized boolean markAsUnused(String item)
{
    for (int i = 0; i < MAX_AVAILABLE; ++i)
    {
        if (item == items[i])
        {
            if (used[i])
            {
                used[i] = false;
                return true;
            }
            else
                return false;
        }
    }
    return false;
}
Listing 10-5’s main thread creates a resource pool, a runnable for repeatedly acquiring and putting back resources, and an array of executors. Each executor is told to execute the runnable.

Pool’s String getItem() and void putItem(String item) methods obtain and return string-based resources. Before obtaining an item in getItem(), the calling thread must acquire a permit from the semaphore, which guarantees that an item is available for use. When the thread finishes with the item, it calls putItem(String), which returns the item to the pool and then releases a permit to the semaphore, which lets another thread acquire that item.

No synchronization lock is held when acquire() is called because that would prevent an item from being returned to the pool. However, String getNextAvailableItem() and boolean markAsUnused(String item) are synchronized to maintain pool consistency. (The semaphore encapsulates the synchronization for restricting access to the pool separately from the synchronization that’s required for maintaining pool consistency.)

Compile Listing 10-5:

javac SemaphoreDemo.java

When you run this application (java SemaphoreDemo), you’ll observe a prefix of the output similar to the following:

```
pool-2-thread-1 acquiring I0
pool-4-thread-1 acquiring I1
pool-6-thread-1 acquiring I2
pool-8-thread-1 acquiring I3
pool-10-thread-1 acquiring I4
pool-1-thread-1 acquiring I5
pool-3-thread-1 acquiring I6
pool-5-thread-1 acquiring I7
pool-7-thread-1 acquiring I8
pool-9-thread-1 acquiring I9
pool-6-thread-1 putting back I2
pool-11-thread-1 acquiring I2
pool-2-thread-1 putting back I0
pool-6-thread-1 acquiring I0
pool-4-thread-1 putting back I1
pool-2-thread-1 acquiring I1
pool-1-thread-1 putting back I5
pool-4-thread-1 acquiring I5
pool-5-thread-1 putting back I7
pool-1-thread-1 acquiring I7
pool-8-thread-1 putting back I3
pool-5-thread-1 acquiring I3
pool-7-thread-1 putting back I8
pool-8-thread-1 acquiring I8
pool-10-thread-1 putting back I4
pool-7-thread-1 acquiring I4
pool-9-thread-1 putting back I9
pool-10-thread-1 acquiring I9
```
Exploring the Concurrent Collections

In Chapter 9, I introduced you to the Collections Framework. This framework provides interfaces and classes that are located in the java.util package. Interfaces include List, Set, and Map; classes include ArrayList, TreeSet, and HashMap.

ArrayList, TreeSet, HashMap, and other implementation classes are not thread-safe. However, you can make them thread-safe by using the synchronized wrapper methods located in the java.util.Collections class. For example, you can pass an ArrayList instance to Collections.synchronizedList() to obtain a thread-safe variant of ArrayList.

Although they’re often needed to simplify code in a multithreaded environment, there are a couple of problems with thread-safe collections:

- It’s necessary to acquire a lock before iterating over a collection that might be modified by another thread during the iteration. If a lock isn’t acquired and the collection is modified, it’s highly likely that java.util.ConcurrentModificationException will be thrown. This happens because Collections Framework classes return fail-fast iterators, which are iterators that throw ConcurrentModificationException when collections are modified during iteration. Fail-fast iterators are often inconvenient to concurrent applications.

- Performance suffers when synchronized collections are accessed frequently from multiple threads. This performance problem ultimately impacts an application’s scalability.

The Concurrency Utilities framework addresses these problems by introducing performant and highly scalable collections-oriented types, which are stored in the java.util.concurrent package. Its collections-oriented classes return weakly consistent iterators, which are iterators that have the following properties:

- An element that’s removed after iteration starts but hasn’t yet been returned via the iterator’s next() method won’t be returned.
- An element that’s added after iteration starts may or may not be returned.
- No element is returned more than once during the collection’s iteration, regardless of changes made to the collection during iteration.

The following list offers a short sample of concurrency-oriented collection types that you’ll find in the java.util.concurrent package:

- BlockingQueue is a subinterface of java.util.Queue that also supports blocking operations that wait for the queue to become nonempty before retrieving an element and wait for space to become available in the queue before storing an element. Each of the ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue,PriorityBlockingQueue, and SynchronousQueue classes implements this interface.

- ConcurrentMap is a subinterface of java.util.Map that declares additional atomic putIfAbsent(), remove(), and replace() methods. The ConcurrentHashMap class (the concurrent equivalent of java.util.HashMap), the ConcurrentNavigableMap class, and the ConcurrentSkipListMap class implement this interface.
Oracle’s Javadoc for BlockingQueue, ArrayBlockingQueue, and other concurrency-oriented collection types identifies these types as part of the Collections Framework.

### Demonstrating BlockingQueue and ArrayBlockingQueue

BlockingQueue’s Javadoc reveals the heart of a producer-consumer application that’s vastly simpler than the equivalent application shown in Chapter 7’s Listing 7-23 because it doesn’t have to deal with synchronization. Listing 10-6 uses BlockingQueue and its ArrayBlockingQueue implementation class in a high-level producer-consumer equivalent.

**Listing 10-6. The Blocking Queue Equivalent of Listing 7-23’s PC Application**

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PC {
    public static void main(String[] args) {
        BlockingQueue<Character> bq;
        final BlockingQueue<Character> bq = new ArrayBlockingQueue<Character>(26);
        final ExecutorService executor = Executors.newFixedThreadPool(2);
        Runnable producer;
        producer = new Runnable() {
            @Override
            public void run() {
                try {
                    for (char ch = 'A'; ch <= 'Z'; ch++)
                        System.out.println(ch + " produced by producer.");
                } catch (InterruptedException ie) {
                    assert false;
                }
            }
        };
    }
```
Listing 10-6 uses BlockingQueue’s put() and take() methods, respectively, to put an object on the blocking queue and to remove an object from the blocking queue. put() blocks when there’s no room to put an object; take() blocks when the queue is empty.

Although BlockingQueue ensures that a character is never consumed before it’s produced, this application’s output may indicate otherwise. For example, here’s a portion of the output from one run:

Y consumed by consumer.
Y produced by producer.
Z consumed by consumer.
Z produced by producer.

Chapter 7’s PC application in Listing 7-24 overcame this incorrect output order by introducing an extra layer of synchronization around setSharedChar() / System.out.println() and an extra layer of synchronization around getSharedChar() / System.out.println(). Later in this chapter, I will show you an alternative in the form of locks.
Learning More About ConcurrentHashMap

Let’s now consider ConcurrentHashMap. This class behaves like HashMap but has been designed to work in multithreaded contexts without the need for explicit synchronization. For example, you often need to check if a map contains a specific value and, when this value is absent, put this value into the map:

```java
if (!map.containsKey("some string-based key"))
    map.put("some string-based key", "some string-based value");
```

Although this code is simple and appears to do the job, it isn’t thread-safe. Between the call to `map.containsKey()` and `map.put()`, another thread could insert this entry, which would then be overwritten. To fix this problem, you would need to explicitly synchronize this code, which I demonstrate here:

```java
synchronized(map)
{
    if (!map.containsKey("some string-based key"))
        map.put("some string-based key", "some string-based value");
}
```

The problem with this approach is that you’ve locked the entire map for read and write operations while checking for key existence and adding the entry to the map when the key doesn’t exist. This locking affects performance when many threads are trying to access the map.

The generic ConcurrentHashMap<V> class addresses this problem by providing the V putIfAbsent(K key, V value) method, which introduces a key/value entry into the map when key is absent. This method is equivalent to the following code fragment but offers better performance:

```java
synchronized(map)
{
    if (!map.containsKey(key))
        return map.put(key, value);
    else
        return map.get(key);
}
```

Using putIfAbsent(), the earlier code fragment translates into the following simpler code fragment:

```java
map.putIfAbsent("some string-based key", "some string-based value");
```

Exploring the Locking Framework

The java.util.concurrent.locks package provides a framework of interfaces and classes for locking and waiting for conditions in a manner that’s distinct from built-in synchronization and java.lang.Object’s wait/notification mechanism. The Locking Framework improves on synchronization and wait/notification by offering capabilities such as lock polling and timed waits.
Java supports synchronization so that threads can safely update shared variables and ensure that a thread's updates are visible to other threads. You leverage synchronization in your code by marking methods or code blocks with the synchronized keyword. The Java virtual machine (JVM) supports synchronization via monitors and the associated monitorenter and monitorexit JVM instructions.

Every Java object is associated with a monitor, which is a mutual exclusion (letting only one thread at a time execute in a critical section) construct that prevents multiple threads from concurrently executing in a critical section. Before a thread can enter a critical section, it's required to lock the monitor. If the monitor is already locked, the thread blocks until the monitor is unlocked (by another thread leaving the critical section).

When a thread locks a monitor, the values of shared variables that are stored in main memory are read into the copies of these variables that are stored in a thread's working memory (also known as local memory or cache memory). This action ensures that the thread will work with the most recent values of these variables and not stale values, and it is known as visibility. The thread proceeds to work with its copies of these shared variables. When the thread unlocks the monitor while leaving the critical section, the values in its copies of shared variables are written back to main memory, which lets the next thread that enters the critical section access the most recent values of these variables. (The volatile keyword addresses visibility only.)

The Locking Framework includes the often-used Lock, ReentrantLock, Condition, ReadWriteLock, and ReentrantReadWriteLock types, which I discuss in this section.

**Lock**

The Lock interface offers more extensive locking operations than can be obtained via the locks associated with monitors. For example, you can immediately back out of a lock-acquisition attempt when a lock isn't available. This interface declares the following methods:

- `void lock()`: Acquires the lock. When the lock isn't available, the calling thread is forced to wait until it becomes available.
- `void lockInterruptibly()`: Acquires the lock unless the calling thread is interrupted. When the lock isn't available, the calling thread is forced to wait until it becomes available or the thread is interrupted, which results in this method throwing InterruptedException.
- `Condition newCondition()`: Returns a new Condition instance that's bound to this Lock instance. This method throws java.lang.UnsupportedOperationException when the Lock implementation doesn't support conditions.
- `boolean tryLock()`: Acquires the lock when it's available at the time this method is invoked. The method returns true when the lock is acquired and false when the lock isn't acquired.
boolean tryLock(long time, TimeUnit unit): Acquires the lock when it's available within the specified waiting time and the calling thread isn't interrupted. When the lock isn't available, the calling thread is forced to wait until it becomes available within the waiting time or the thread is interrupted, which results in this method throwing InterruptedException.

void unlock(): Releases the lock.

Acquired locks must be released. In the context of synchronized methods and statements, and the implicit monitor lock associated with every object, all lock acquisition and release occurs in a block-structured manner. When multiple locks are acquired, they're released in the opposite order and all locks are released in the same lexical scope in which they were acquired.

Lock acquisition and release in the context of Lock interface implementations can be more flexible. For example, some algorithms for traversing concurrently accessed data structures require the use of hand-over-hand or chain locking: you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on. Implementations of the Lock interface enable the use of such techniques by allowing a lock to be acquired and released in different scopes, and by allowing multiple locks to be acquired and released in any order.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. As a result, you should typically employ the following idiom for lock acquisition and release:

```java
Lock l = ...; // ... is a placeholder for code that obtains the lock
l.lock();
try
{
    // access the resource protected by this lock
} catch (Exception ex)
{
    // restore invariants
} finally
{
    l.unlock();
}
```

This idiom ensures that an acquired lock will always be released.

**Note** All Lock implementations are required to enforce the same memory synchronization semantics as provided by the built-in monitor lock.
ReentrantLock

Lock is implemented by the ReentrantLock class, which describes a reentrant mutual exclusion lock. This lock is associated with a hold count. When a thread holds the lock and reacquires the lock by invoking lock(), lockUninterruptibly(), or one of the tryLock() methods, the hold count is increased by 1. When the thread invokes unlock(), the hold count is decremented by 1. The lock is released when this count reaches 0.

ReentrantLock offers the same concurrency and memory semantics as the implicit monitor lock that’s accessed via synchronized methods and statements. However, it has extended capabilities and offers better performance under high thread contention (threads frequently asking to acquire a lock that’s already held by another thread). When many threads attempt to access a shared resource, the JVM spends less time scheduling these threads and more time executing them.

You initialize a ReentrantLock instance by invoking either of the following constructors:

- ReentrantLock(): Creates an instance of ReentrantLock. This constructor is equivalent to ReentrantLock(false).
- ReentrantLock(boolean fair): Creates an instance of ReentrantLock with the specified fairness policy. Pass true to fair when this lock should use a fair ordering policy: under contention, the lock would favor granting access to the longest-waiting thread.

ReentrantLock implements Lock’s methods. However, its implementation of unlock() throws java.lang.IllegalMonitorStateException when the calling thread doesn’t hold the lock. Also, ReentrantLock provides its own methods. For example, boolean isFair() returns the fairness policy and boolean isHeldByCurrentThread() returns true when the lock is held by the current thread.

Listing 10-7 demonstrates Lock and ReentrantLock in a variation of Listing 10-6 that ensures that the output is never shown in incorrect order (a consumed message appearing before a produced message). (This listing is the Concurrency Utilities counterpart to Chapter 7’s Listing 7-24.)

Listing 10-7. Achieving Synchronization in Terms of Locks

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PC
{
    public static void main(String[] args)
    {
        final Lock lock = new ReentrantLock();
        final BlockingQueue<Character> bq;
        bq = new ArrayBlockingQueue<Character>(26);
        final ExecutorService executor = Executors.newFixedThreadPool(2);
        Runnable producer;
```
CHAPTER 10: Exploring the Concurrency Utilities

```
producer = new Runnable()
{
    @Override
    public void run()
    {
        for (char ch = 'A'; ch <= 'Z'; ch++)
        {
            try
            {
                lock.lock();
                try
                {
                    while (!bq.offer(ch))
                    {
                        lock.unlock();
                        Thread.sleep(50);
                        lock.lock();
                    }
                    System.out.println(ch + " produced by producer.");
                }
                catch (InterruptedException ie)
                {
                    assert false;
                }
            }
            finally
            {
                lock.unlock();
            }
        }
    }
}
executor.execute(producer);
Runnable consumer;
consumer = new Runnable()
{
    @Override
    public void run()
    {
        char ch = '\0';
        do
        {
            try
            {
                lock.lock();
                try
                {

```
Character c;
while ((c = bq.poll()) == null)
{
    lock.unlock();
    Thread.sleep(50);
    lock.lock();
}
ch = c; // unboxing behind the scenes
System.out.println(ch + " consumed by consumer.");
}
catch (InterruptedException ie)
{
    assert false;
}
finally
{
    lock.unlock();
}

while (ch != 'Z');
executor.shutdownNow();
}
executor.execute(consumer);
}

Listing 10-7 uses Lock's lock() and unlock() methods to obtain and release a lock. When a thread calls lock() and the lock is unavailable, the thread is disabled (and cannot be scheduled) until the lock becomes available.

This listing also uses BlockingQueue's offer() method instead of put() to store an object in the blocking queue and its poll() method instead of take() to retrieve an object from the queue. These alternative methods are used because they don't block.

If I had used put() and take(), this application would have deadlocked in the following scenario:

1. The consumer thread acquires the lock via its lock.lock() call.
2. The producer thread attempts to acquire the lock via its lock.lock() call and is disabled because the consumer thread has already acquired the lock.
3. The consumer thread calls take() to obtain the next java.lang.Character object from the queue.
4. Because the queue is empty, the consumer thread must wait.
5. The consumer thread doesn't give up the lock that the producer thread requires before waiting, so the producer thread also continues to wait.
Compile this source code as follows:

```java
cjavac PC.java
```

When you run this application (java PC), you’ll discover that, as with Listing 7-24’s PC application, it never outputs a consuming message before a producing message for the same item.

## Condition

The Condition interface factors out Object’s wait and notification methods (wait(), notify(), and notifyAll()) into distinct condition objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where Lock replaces synchronized methods and statements, Condition replaces Object’s wait/notification methods.

**Note**

A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular Lock instance, use Lock’s newCondition() method.

Condition declares the following methods:

- `void await()`: Forces the calling thread to wait until it’s signaled or interrupted.
- `boolean await(long time, TimeUnit unit)`: Forces the calling thread to wait until it’s signaled or interrupted, or until the specified waiting time elapses.
- `long awaitNanos(long nanosTimeout)`: Forces the current thread to wait until it’s signaled or interrupted, or until the specified waiting time elapses.
- `void awaitUninterruptibly()`: Forces the current thread to wait until it’s signaled.
- `boolean awaitUntil(Date deadline)`: Forces the current thread to wait until it’s signaled or interrupted, or until the specified deadline elapses.
- `void signal()`: Wakes up one waiting thread.
- `void signalAll()`: Wakes up all waiting threads.

Listing 10-8 revisits the producer-consumer example to show you how it can be written to take advantage of conditions.

### Listing 10-8. Achieving Synchronization in Terms of Locks and Conditions

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```
public class PC
{
    public static void main(String[] args)
    {
        Shared s = new Shared();
        new Producer(s).start();
        new Consumer(s).start();
    }
}

class Shared
{
    // Fields c and available are volatile so that writes to them are visible to
    // the various threads. Fields lock and condition are final so that they're
    // initial values are visible to the various threads. (The Java memory model
    // promises that, after a final field has been initialized, any thread will
    // see the same [correct] value.)

    private volatile char c;
    private volatile boolean available;
    private final Lock lock;
    private final Condition condition;

    Shared()
    {
        c = '\u0000';
        available = false;
        lock = new ReentrantLock();
        condition = lock.newCondition();
    }

    Lock getLock()
    {
        return lock;
    }

    char getSharedChar()
    {
        lock.lock();
        try
        {
            while (!available)
            {
                available = false;
            }
        }
    }
}
catch (InterruptedException ie)
{
    ie.printStackTrace();
}
available = false;
condition.signal();
}
finally
{
    lock.unlock();
    return c;
}
}

void setSharedChar(char c)
{
    lock.lock();
    try
    {
        while (available)
        {
            try
            {
                condition.await();
            }
            catch (InterruptedException ie)
            {
                ie.printStackTrace();
            }
        }
        this.c = c;
        available = true;
        condition.signal();
    }
    finally
    {
        lock.unlock();
    }
}

class Producer extends Thread
{
    // l is final because it's initialized on the main thread and accessed on the
    // producer thread.
    private final Lock l;

    // s is final because it's initialized on the main thread and accessed on the
    // producer thread.
    private final Shared s;
Producer(Shared s)
{
    this.s = s;
    l = s.getLock();
}

@Override
public void run()
{
    for (char ch = 'A'; ch <= 'Z'; ch++)
    {
        l.lock();
        s.setSharedChar(ch);
        System.out.println(ch + " produced by producer.");
        l.unlock();
    }
}

class Consumer extends Thread
{
    // l is final because it's initialized on the main thread and accessed on the
    // consumer thread.
    private final Lock l;

    // s is final because it's initialized on the main thread and accessed on the
    // consumer thread.
    private final Shared s;

    Consumer(Shared s)
    {
        this.s = s;
        l = s.getLock();
    }

    @Override
    public void run()
    {
        char ch;
        do
        {
            l.lock();
            ch = s.getSharedChar();
            System.out.println(ch + " consumed by consumer.");
            l.unlock();
        } while (ch != 'Z');
    }
}
Listing 10-8 is similar to Listing 7-24's PC application. However, it replaces synchronized and wait/notification with locks and conditions.

PC's main() method instantiates the Shared, Producer, and Consumer classes. The Shared instance is passed to the Producer and Consumer constructors, and these threads are then started.

The Producer and Consumer constructors are called on the main thread. Because the producer and consumer threads also access the Shared instance, this instance must be visible to these threads (especially when these threads run on different cores). In each of Producer and Consumer, I accomplish this task by declaring s to be final. I could have declared this field to be volatile, but volatile suggests additional writes to the field and s shouldn't be changed after being initialized.

Check out Shared's constructor. Notice that it creates a lock via lock = new ReentrantLock();, and creates a condition associated with this lock via condition = lock.newCondition();. This lock is made available to the producer and consumer threads via the Lock getLock() method.

The producer thread invokes Shared's void setSharedChar(char c) method to generate a new character and then outputs a message identifying the produced character. This method locks the previously created Lock object and enters a while loop that repeatedly tests variable available, which is true when a produced character is available for consumption.

While available is true, the producer invokes the condition's await() method to wait for available to become false. The consumer signals the condition to wake up the producer when it has consumed the character. (I use a loop instead of an if statement because spurious wakeups are possible and available might still be true.)

After leaving its loop, the producer thread records the new character, assigns true to available to indicate that a new character is available for consumption, and signals the condition to wake up a waiting consumer. Lastly, it unlocks the lock and exits setSharedChar().

Note I lock the setSharedChar()//System.out.println() block in Producer's run() method and the getSharedChar()//System.out.println() block in Consumer's run() method to prevent the application from outputting consuming messages before producing messages, even though characters are produced before they're consumed.

The behavior of the consumer thread and getSharedChar() method is similar to what I've just described for the producer thread and setSharedChar() method.

Note I didn't use the try/finally idiom for ensuring that a lock is disposed of in Producer's and Consumer's run() methods, because an exception isn't thrown from this context.

Compile this source code as follows:

javac PC.java
Run this application (java PC) and you'll observe output identical to the following prefix of the output, which indicates lockstep synchronization (the producer thread doesn’t produce an item until it’s consumed and the consumer thread doesn’t consume an item until it’s produced):

<table>
<thead>
<tr>
<th></th>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>produced by</td>
<td>consumed by</td>
</tr>
<tr>
<td></td>
<td>producer.</td>
<td>consumer.</td>
</tr>
<tr>
<td>B</td>
<td>produced by</td>
<td>consumed by</td>
</tr>
<tr>
<td></td>
<td>producer.</td>
<td>consumer.</td>
</tr>
<tr>
<td>C</td>
<td>produced by</td>
<td>consumed by</td>
</tr>
<tr>
<td></td>
<td>producer.</td>
<td>consumer.</td>
</tr>
<tr>
<td>D</td>
<td>produced by</td>
<td>consumed by</td>
</tr>
<tr>
<td></td>
<td>producer.</td>
<td>consumer.</td>
</tr>
</tbody>
</table>

**ReadWriteLock**

Situations arise where data structures are read more often than they’re modified. For example, you may have created an online dictionary of word definitions that many threads will read concurrently, whereas a single thread may add new definitions or update existing definitions. The Locking Framework provides a read-write locking mechanism for these situations that yields greater concurrency when reading and the safety of exclusive access when writing. This mechanism is based on the ReadWriteLock interface.

ReadWriteLock maintains a pair of locks: one lock for read-only operations and one lock for write operations. Multiple reader threads may hold the read lock simultaneously, as long as there are no writers. The write lock is exclusive: only a single thread can modify shared data. (The lock that’s associated with the synchronized keyword is also exclusive.)

ReadWriteLock declares the following methods:

- Lock readLock(): Returns the lock that’s used for reading.
- Lock writeLock(): Returns the lock that’s used for writing.

**ReentrantReadWriteLock**

ReadWriteLock is implemented by the ReentrantReadWriteLock class, which describes a reentrant read-write lock with similar semantics to ReentrantLock.

You initialize a ReentrantReadWriteLock instance by invoking either of the following constructors:

- ReentrantReadWriteLock(): Creates an instance of ReentrantReadWriteLock. This constructor is equivalent to ReentrantReadWriteLock(false).
- ReentrantReadWriteLock(boolean fair): Creates an instance of ReentrantReadWriteLock with the specified fairness policy. Pass true to fair when this lock should use a fair ordering policy.
Note For the fair ordering policy, when the currently held lock is released, either the longest-waiting single writer thread will be assigned the write lock or, when there’s a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock.

A thread that tries to acquire a fair read lock (non-reentrantly) will block when the write lock is held or there’s a waiting writer thread. The thread will not acquire the read lock until after the oldest currently waiting writer thread has acquired and released the write lock. If a waiting writer abandons its wait, leaving one or more reader threads as the longest waiters in the queue with the write lock free, those readers will be assigned the read lock.

A thread that tries to acquire a fair write lock (non-reentrantly) will block unless both the read lock and write lock are free (which implies no waiting threads). (The nonblocking tryLock() methods don’t honor this fair setting and will immediately acquire the lock if possible, regardless of waiting threads.)

After instantiating this class, you would invoke the following methods to obtain the read and write locks:

- ReentrantReadWriteLock.ReadLock readLock(): Returns the lock used for reading.
- ReentrantReadWriteLock.WriteLock writeLock(): Returns the lock used for writing.

Each of the nested ReadLock and WriteLock classes implements the Lock interface and declares its own methods. Furthermore, ReentrantReadWriteLock declares additional methods such as the following pair:

- int getReadHoldCount(): Returns the number of reentrant read holds on this lock by the calling thread, which is 0 when the read lock isn’t held by the calling thread. A reader thread has a hold on a lock for each lock action that’s not matched by an unlock action.
- int getWriteHoldCount(): Returns the number of reentrant write holds on this lock by the calling thread, which is 0 when the write lock isn’t held by the calling thread. A writer thread has a hold on a lock for each lock action that’s not matched by an unlock action.

To demonstrate ReadWriteLock and ReentrantReadWriteLock, Listing 10-9 presents an application whose writer thread populates a dictionary of word/definition entries while a reader thread continually accesses entries at random and outputs them.

**Listing 10-9. Using ReadWriteLock to Satisfy a Dictionary Application’s Reader and Writer Threads**

```java
import java.util.HashMap;
import java.util.Map;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Dictionary
{
    public static void main(String[] args)
    {
        final String[] words =
        {
            "hypocalcemia",
            "prolixity",
            "assiduous",
            "indefatigable",
            "castellan"
        };

        final String[] definitions =
        {
            "a deficiency of calcium in the blood",
            "unduly prolonged or drawn out",
            "showing great care, attention, and effort",
            "able to work or continue for a lengthy time without tiring",
            "the govenor or warden of a castle or fort"
        };

        final Map<String, String> dictionary = new HashMap<String, String>();

        ReadWriteLock rwl = new ReentrantReadWriteLock(true);
        final Lock rlock = rwl.readLock();
        final Lock wlock = rwl.writeLock();

        Runnable writer = new Runnable()
        {
            @Override
            public void run()
            {
                for (int i = 0; i < words.length; i++)
                {
                    wlock.lock();
                    try
                    {
                        dictionary.put(words[i], definitions[i]);
                        System.out.println("writer storing " +
                            words[i] + " entry");
                    }
                    finally
                    {
                        wlock.unlock();
                    }
                }
            }
        };
    }
}
try {
    Thread.sleep((int) Math.random() * 500);
} catch (InterruptedException ie) {
    System.err.println("writer interrupted");
}

ExecutorService es = Executors.newFixedThreadPool(1);
es.submit(writer);

Runnable reader = new Runnable()
{
    @Override
    public void run()
    {
        while (true)
        {
            rlock.lock();

            try
            {
                int i = (int) (Math.random() * words.length);
                System.out.println("reader accessing "+
                words[i] + ": "+
                dictionary.get(words[i])
                + " entry");
            }
            finally
            {
                rlock.unlock();
            }
        }
    }
}

es = Executors.newFixedThreadPool(1);
es.submit(reader);
}

Listing 10-9's main thread first creates the words and definitions arrays of strings, which are declared final because they will be accessed from anonymous classes. After creating a map in which to store word/definition entries, it obtains a reentrant read/write lock and accesses the reader and writer locks.

A runnable for the writer thread is now created. Its run() method iterates over the words array. Each of the iterations locks the writer lock. When this method returns, the writer thread has the exclusive writer lock and can update the map. It does so by invoking the map's put() method. After outputting a message to identify the added word, the writer thread releases the lock and sleeps for a random
amount of time to give the appearance of performing other work. An executor based on a thread pool is obtained and used to invoke the writer thread’s runnable.

A runnable for the reader thread is subsequently created. Its `run()` method repeatedly obtains the read lock, accesses a random entry in the map, outputs this entry, and unlocks the read lock. An executor based on a thread pool is obtained and used to invoke the reader thread’s runnable.

Although I could have avoided the idiom for lock acquisition and release because an exception isn’t thrown, I specified `try/finally` for good form.

Compile Listing 10-9 as follows:

```bash
class Dictionary {
    private final Map map = new HashMap();
    private synchronized void store(String key, String value) {
        map.put(key, value);
    }
    private synchronized String access(String key) {
        return map.get(key);
    }
}
```

Run this application (java Dictionary) and you should observe output that’s similar to the following prefix of output that I observed from one execution:

```java
writer storing hypocalcemia entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing indefatigable: null entry
reader accessing indefatigable: null entry
reader accessing castellan: null entry
writer storing prolixity entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
writer storing assiduous entry
reader accessing indefatigable: null entry
reader accessing castellan: null entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
writer storing indefatigable entry
reader accessing indefatigable: able to work or continue for a lengthy time without tiring entry
writer storing castellan entry
reader accessing indefatigable: able to work or continue for a lengthy time without tiring entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing castellan: the governor or warden of a castle or fort entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing prolixity: unduly prolonged or drawn out entry
```

---

**Exploring Atomic Variables**

The `java.util.concurrent.atomic` package provides a small toolkit of classes that support lock-free, thread-safe operations on single variables. The classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update so that external synchronization isn’t required.
Some of the classes located in this package are described below:

- **AtomicBoolean**: A boolean value that may be updated atomically.
- **AtomicInteger**: An int value that may be updated atomically.
- **AtomicIntegerArray**: An int array whose elements may be updated atomically.
- **AtomicLong**: A long value that may be updated atomically.
- **AtomicLongArray**: A long array whose elements may be updated atomically.
- **AtomicReference**: An object reference that may be updated atomically.
- **AtomicReferenceArray**: An object reference array whose elements may be updated atomically.

Listing 7-19 declared a small utility class named ID for returning unique long integer identifiers via ID's `getNextID()` class method. Because this method wasn’t synchronized, multiple threads could obtain the same identifier. Listing 10-10 fixes this problem by including reserved word `synchronized` in the method header.

**Listing 10-10. Returning Unique Identifiers in a Thread-Safe Manner via synchronized**

```java
class ID {
    private static long nextID = 0;
    static synchronized long getNextID() {
        return nextID++;
    }
}
```

Although `synchronized` is appropriate for this example, excessive use of this reserved word in more complex classes can lead to deadlock, starvation, or other problems. Listing 10-11 shows you how to avoid these assaults on a concurrent application’s liveness (the ability to execute in a timely manner) by replacing `synchronized` with an atomic variable.

**Listing 10-11. Returning Unique IDs in a Thread-Safe Manner via AtomicLong**

```java
import java.util.concurrent.atomic.AtomicLong;

class ID {
    private static AtomicLong nextID = new AtomicLong(0);
    static long getNextID() {
        return nextID.getAndIncrement();
    }
}
```
In Listing 10-11, I've converted `nextID` from a `long` to an `AtomicLong` instance, initializing this object to 0. I've also refactored the `getNextID()` method to call `AtomicLong`'s `getAndIncrement()` method, which increments the `AtomicLong` instance's internal long integer variable by 1 and returns the previous value in one indivisible step.

### Improving Performance with the Concurrency Utilities

Java's low-level synchronization mechanism, which enforces *mutual exclusion* (the thread holding the lock that guards a set of variables has exclusive access to them) and *visibility* (changes to the guarded variables become visible to other threads that subsequently acquire the lock), impacts hardware utilization and scalability in the following ways:

- **Contended synchronization** (multiple threads constantly competing for a lock) is expensive and throughput suffers as a result. This expense is caused mainly by the frequent *context switching* that occurs. Each context switch operation can take many processor cycles to complete. In contrast, modern JVMs make *uncontended synchronization* inexpensive.

- When a thread holding a lock is delayed (because of a scheduling delay, for example), no thread that requires that lock makes any progress; the hardware isn't utilized as well as it might be.

Although you might believe that you can use `volatile` as a synchronization alternative, this won’t work. Volatile variables only solve the visibility problem. They cannot be used to implement safely the atomic read-modify-write sequences that are necessary for implementing thread-safe counters and other entities that require mutual exclusion. However, there is an alternative that’s responsible for the performance gains offered by the Concurrency Utilities over equivalent features in the Threads API. This alternative is known as Compare-and-Swap.

**Compare-and-Swap (CAS)** is the generic term for an uninterruptible microprocessor-specific instruction that reads a memory location, compares the read value with an expected value, and stores a new value in the memory location when the read value matches the expected value. Otherwise, nothing is done. Modern microprocessors offer variations of CAS. For example, Intel microprocessors provide the `cmpxchg` family of instructions, whereas the older PowerPC microprocessors provide equivalent load-link (such as `lwarx`) and store-conditional (such as `stwcx`) instructions.

CAS supports atomic read-modify-write sequences. You typically use CAS as follows:

1. Read value `x` from address `A`.
2. Perform a multistep computation on `x` to derive a new value `y`.
3. Use CAS to change the value of `A` from `x` to `y`. CAS succeeds when `A`'s value hasn’t changed while performing these steps.

To understand CAS’s benefit, consider Listing 10-10’s `ID` class, which returns a unique identifier. Because this class declares its `getNextID()` method synchronized, high contention for the monitor lock results in excessive context switching that can delay all of the threads and result in an application that doesn’t scale well.
Assume the existence of a CAS class that stores an int-based value in value. Furthermore, it offers atomic methods int getValue() for returning value and int compareAndSwap(int expectedValue, int newValue) for implementing CAS. Behind the scenes, CAS relies on the Java Native Interface (see Chapter 16) to access the microprocessor-specific CAS instruction.

The compareAndSwap() method executes the following instruction sequence atomically:

```java
int readValue = value; // Obtain the stored value.
if (readValue == expectedValue) // If stored value not modified ...
    value = newValue; // ... change to new value.
return readValue; // Return value before a potential change.
```

Listing 10-12 presents a new version of ID that uses the CAS class to obtain a unique identifier in a highly performant manner.

**Listing 10-12. Returning Unique IDs in a Thread-Safe Manner via CAS**

class ID
{
    private static CAS value = new CAS(0);

    static long getNextID()
    {
        int curValue = value.getValue();
        while (value.compareAndSwap(curValue, curValue + 1) != curValue)
            curValue = value.getValue();
        return curValue - 1;
    }
}

ID encapsulates a CAS instance initialized to int-value 0 and declares a getNextID() method for retrieving the current identifier value and then incrementing this value with help from this instance. After retrieving the instance’s current value, getNextID() repeatedly invokes compareAndSwap() until curValue’s value hasn’t changed (by another thread). This method is then free to change this value, after which it returns the previous value. When no lock is involved, contention is avoided along with excessive context switching. Performance improves and the code is more scalable.

As an example of how CAS improves the Concurrency Utilities, consider ReentrantLock. This class offers better performance than synchronized under high thread contention. To boost performance, ReentrantLock’s synchronization is managed by a subclass of the abstract java.util.concurrent.locks.AbstractQueuedSynchronizer class. In turn, this class leverages the undocumented sun.misc.Unsafe class and its compareAndSwapInt() CAS method.

The atomic variable classes also leverage CAS. Furthermore, they provide a method that has the following form:

```java
boolean compareAndSet(expectedValue, updateValue)
```

This method (which varies in argument types across different classes) atomically sets a variable to the updateValue when it currently holds the expectedValue, reporting true on success.
CHAPTER 10: Exploring the Concurrency Utilities

EXERCISES

The following exercises are designed to test your understanding of Chapter 10’s content.

1. Define Concurrency Utilities.
2. Identify the packages in which Concurrency Utilities types are stored.
3. Define task.
4. Define executor.
5. Identify the Executor interface’s limitations.
6. How are Executor’s limitations overcome?
7. What differences exist between Runnable’s run() method and Callable’s call() method?
8. True or false: You can throw checked and unchecked exceptions from Runnable’s run() method but can only throw unchecked exceptions from Callable’s call() method.
10. Describe the Executors class’s newFixedThreadPool() method.
11. Define synchronizer.
12. Identify and describe four commonly used synchronizers.
13. What concurrency-oriented extensions to the Collections Framework are provided by the Concurrency Utilities?
14. Define lock.
15. What is the biggest advantage that Lock objects hold over the implicit locks that are obtained when threads enter critical sections (controlled via the synchronized reserved word)?
16. How do you obtain a Condition instance for use with a particular Lock instance?
17. Define atomic variable.
18. What does the AtomicIntegerArray class describe?
19. True or false: volatile supports atomic read-modify-write sequences.
20. What’s responsible for the performance gains offered by the Concurrency Utilities?
21. Listing 7-13 in Chapter 7 presented a simple CountingThreads application. Refactor this application to work with Executors and ExecutorService.
22. When you execute the previous exercise’s CountingThreads application, you’ll observe output that identifies the threads via names such as pool-1-thread-1. Modify CountingThreads so that you observe names A and B. Hint: You’ll need to use ThreadFactory.
23. Listing 7-25’s DeadlockDemo class demonstrates deadlock in the context of the synchronized primitive. Create an equivalent application that demonstrates deadlock via the Lock interface and ReentrantLock class. Then create an application that uses these same types to prevent deadlock.
24. Convert the following expressions to their atomic variable equivalents:
   
   ```java
   int total = ++counter;
   int total = counter--;
   ```
Summary

The low-level Threads API lets you create multithreaded applications that offer better performance and responsiveness over their single-threaded counterparts. However, performance issues that affect an application’s scalability and other problems resulted in Java 5’s introduction of the Concurrency Utilities.

The Concurrency Utilities organizes its many types into three packages: java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks. Basic types, such as executors, thread pools, and concurrent hashmaps, are stored in java.util.concurrent, classes that support lock-free, thread-safe programming on single variables are stored in java.util.concurrent.atomic, and types for locking and waiting on conditions are stored in java.util.concurrent.locks.

An executor decouples task submission from task-execution mechanics and is described by the Executor, ExecutorService, and ScheduledExecutorService interfaces. You obtain an executor by calling one of the utility methods in the Executors class. Executors are associated with callables and futures.

A synchronizer facilitates common forms of synchronization. Countdown latches, cyclic barriers, exchangers, and semaphores are commonly used synchronizers.

A concurrent collection is an extension to the Collections Framework. The BlockingQueue and ConcurrentMap interfaces along with the ArrayBlockingQueue and ConcurrentHashMap classes are examples.

The java.util.concurrent.locks package provides a framework of interfaces and classes for locking and waiting for conditions in a manner that’s distinct from built-in synchronization and Object’s wait/notify mechanism. Java supports locks via the commonly used Lock, Condition, and ReadWriteLock interfaces; and via the ReentrantLock and ReentrantReadWriteLock classes.

The java.util.concurrent.atomic package provides a small toolkit of classes that support lock-free, thread-safe operations on single variables. The classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update so that external synchronization isn’t required. Examples of atomic variable classes include AtomicBoolean and AtomicIntegerArray.

Java’s synchronization mechanism impacts hardware utilization and scalability. To overcome this problem, the various Concurrency Utilities types (including the atomic variable classes) are based on the Compare-and-Swap (CAS) instruction, which offers better performance.

CAS is the generic term for an uninterruptible microprocessor-specific instruction that reads a memory location, compares the read value with an expected value, and stores a new value in the memory location when the read value matches the expected value. Otherwise, nothing is done. Modern microprocessors offer variations of CAS. For example, Intel microprocessors provide the cmpxchg family of instructions, whereas older PowerPC microprocessors provide equivalent load-link (such as lwarx) and store-conditional (such as stwcx) instructions.

This chapter ends our tour of Java’s Collections Framework and related Concurrency Utilities. In Chapter 11, we’ll explore Java’s classic I/O APIs: File, RandomAccessFile, streams, and writers/readers.
Performing Classic I/O

Applications often input data for processing and output processing results. Data is input from a file or some other source and is output to a file or some other destination. Java supports I/O via the classic I/O APIs located in the java.io package and the new I/O APIs located in java.nio and related subpackages (and java.util.regex). This chapter introduces you to the classic I/O APIs.

Note You’ve already experienced classic I/O in the context of Chapter 1’s Standard I/O coverage.

Working with the File API

Applications often interact with a filesystem, which is usually expressed as a hierarchy of files and directories starting from a root directory.

Android and other platforms on which a virtual machine runs typically support at least one filesystem. For example, a Unix/Linux (and Linux-based Android) platform combines all mounted (attached and prepared) disks into one virtual filesystem. In contrast, Windows associates a separate filesystem with each active disk drive.

Java offers access to the underlying platform’s available filesystem(s) via its concrete java.io.File class. File declares the File[] listRoots() class method to return the root directories (roots) of available filesystems as an array of File objects.

Note The set of available filesystem roots is affected by platform-level operations, such as inserting or ejecting removable media, and disconnecting or unmounting physical or virtual disk drives.

Listing 11-1 presents a DumpRoots application that uses listRoots() to obtain an array of available filesystem roots and then outputs the array’s contents.
Listing 11-1. Dumping Available Filesystem Roots to Standard Output

```java
import java.io.File;

public class DumpRoots {
    public static void main(String[] args) {
        File[] roots = File.listRoots();
        for (File root : roots) {
            System.out.println(root);
        }
    }
}
```

When I run this application on my Windows 7 platform, I receive the following output, which reveals four available roots:

```
C:\
D:\
E:\
F:\
```

If I happened to run DumpRoots on a Unix or Linux platform, I would receive one line of output that consists of the virtual filesystem root (`/`).

**Constructing File Instances**

Apart from using `listRoots()`, you can obtain a `File` instance by calling a `File` constructor such as `File(String pathname)`, which creates a `File` instance that stores the `pathname` string. The following assignment statements demonstrate this constructor:

```java
File file1 = new File("/x/y");
File file2 = new File("C:\temp\x.dat");
```

The first statement assumes a Unix/Linux platform, starts the `pathname` with root directory symbol `/`, and continues with directory name `x`, separator character `/`, and file or directory name `y`. (It also works on Windows, which assumes this path begins at the root directory on the current drive.)

The second statement assumes a Windows platform, starts the `pathname` with drive specifier `C:`, and continues with root directory symbol `\`, directory name `temp`, separator character `\`, and filename `x.dat` (although `x.dat` might refer to a directory). (I could also use forward slashes on Windows.)

**Note** A *path* is a hierarchy of directories that must be traversed to locate a file or a directory. A *pathname* is a string representation of a path; a platform-dependent *separator character* (such as the Windows backslash `\` character) appears between consecutive names.
CHAPTER 11: Performing Classic I/O

Caution Always double backslash characters that appear in a string literal, especially when specifying a pathname; otherwise, you run the risk of bugs or compiler error messages. For example, I doubled the backslash characters in the second statement to denote a backslash and not a tab (\t) and to avoid a compiler error message (\x is illegal).

Each statement’s pathname is an absolute pathname, which is a pathname that starts with the root directory symbol; no other information is required to locate the file/directory that it denotes. In contrast, a relative pathname doesn’t start with the root directory symbol; it’s interpreted via information taken from some other pathname.

Note The java.io package’s classes default to resolving relative pathnames against the current user (also known as working) directory, which is identified by system property user.dir and which is typically the directory in which the virtual machine was launched. (Chapter 7 showed you how to read system properties via java.lang.System’s getProperty() method.)

File instances contain abstract representations of file and directory pathnames (these files or directories may or may not exist in their filesystems) by storing abstract pathnames, which offer platform-independent views of hierarchical pathnames. In contrast, user interfaces and operating systems use platform-dependent pathname strings to name files and directories.

An abstract pathname consists of an optional platform-dependent prefix string, such as a disk drive specifier—which is / for the Unix/Linux root directory or \ for a Windows Universal Naming Convention (UNC) pathname—and a sequence of zero or more string names. The first name in an abstract pathname may be a directory name or, in the case of Windows UNC pathnames, a hostname. Each subsequent name denotes a directory; the last name may denote a directory or a file. The empty abstract pathname has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently platform dependent. When a pathname string is converted into an abstract pathname, the names within this string may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying platform. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default name-separator character.

Note The default name-separator character is defined by the system property file.separator and is made available in File's public static separator and separatorChar fields; the first field stores the character in a java.lang.String instance and the second field stores it as a char value.

File offers additional constructors for instantiating this class. For example, the following constructors merge parent and child pathnames into combined pathnames that are stored in File objects:

- File(String parent, String child) creates a new File instance from a parent pathname string and a child pathname string.
- File(File parent, String child) creates a new File instance from a parent pathname File instance and a child pathname string.
Each constructor's parent parameter is passed a *parent pathname*, a string that consists of all pathname components except for the last name, which is specified by child. The following statement demonstrates this concept via `File(String, String)`: 

```java
File file3 = new File("prj/books/", "ljfad3");
```

The constructor merges parent pathname `prj/books/` with child pathname `ljfad3` into pathname `prj/books/ljfads3`. (If I had specified `prj/books` as the parent pathname, the constructor would have added the separator character after `books`.)

**Tip** Because `File(String pathname), File(String parent, String child), and File(File parent, String child)` don't detect invalid pathname arguments (apart from throwing `java.lang.NullPointerException` when `pathname` or `child` is `null`), you must be careful when specifying pathnames. You should strive to only specify pathnames that are valid for all platforms on which the application will run. For example, instead of hard-coding a drive specifier (such as `C:`) in a pathname, use the roots that are returned from `listRoots()`. Even better, keep your pathnames relative to the current user/working directory (returned from the `user.dir` system property).

---

**Learning About Stored Abstract Pathnames**

After obtaining a `File` object, you can interrogate it to learn about its stored abstract pathname by calling the methods that are described in Table 11-1.

**Table 11-1. File Methods for Learning About a Stored Abstract Pathname**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>File getAbsoluteFile()</code></td>
<td>Returns the absolute form of this <code>File</code> object's abstract pathname. This method is equivalent to <code>new File(this.getAbsolutePath())</code>.</td>
</tr>
<tr>
<td><code>String getAbsolutePath()</code></td>
<td>Returns the absolute pathname string of this <code>File</code> object's abstract pathname. When it's already absolute, the pathname string is returned as if by calling <code>getPath()</code>. When it's the empty abstract pathname, the pathname string of the current user directory (identified via <code>user.dir</code>) is returned. Otherwise, the abstract pathname is resolved in a platform-dependent manner. On Unix/Linux platforms, a relative pathname is made absolute by resolving it against the current user directory. On Windows platforms, the pathname is made absolute by resolving it against the current directory of the drive named by the pathname, or the current user directory when there is no drive.</td>
</tr>
<tr>
<td><code>File getCanonicalFile()</code></td>
<td>Returns the <em>canonical</em> (simplest possible, absolute, and unique) form of this <code>File</code> object's abstract pathname. This method throws <code>java.io.IOException</code> when an I/O error occurs (creating the canonical pathname may require filesystem queries); it equates to <code>new File(this.getCanonicalPath())</code>.</td>
</tr>
</tbody>
</table>

(continued)
Table 11-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getCanonicalPath()</td>
<td>Returns the canonical pathname string of this File object’s abstract pathname. This method first converts this pathname to absolute form when necessary, as if by invoking getAbsolutePath(), and then maps it to its unique form in a platform-dependent way. Doing so typically involves removing redundant names such as . and .. from the pathname, resolving symbolic links (on Unix/Linux platforms), and converting drive letters to a standard case (on Windows platforms). This method throws IOException when an I/O error occurs (creating the canonical pathname may require filesystem queries).</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns the filename or directory name denoted by this File object’s abstract pathname. This name is the last in a pathname’s name sequence. The empty string is returned when the pathname’s name sequence is empty.</td>
</tr>
<tr>
<td>String getParent()</td>
<td>Returns the parent pathname string of this File object’s pathname, or returns null when this pathname doesn’t name a parent directory.</td>
</tr>
<tr>
<td>File getParentFile()</td>
<td>Returns a File object storing this File object’s abstract pathname’s parent abstract pathname; returns null when the parent pathname isn’t a directory.</td>
</tr>
<tr>
<td>String getPath()</td>
<td>Converts this File object’s abstract pathname into a pathname string where the names in the sequence are separated by the character stored in File’s separator field. Returns the resulting pathname string.</td>
</tr>
<tr>
<td>boolean isAbsolute()</td>
<td>Returns true when this File object’s abstract pathname is absolute; otherwise, returns false when it’s relative. The definition of absolute pathname is system dependent. On Unix/Linux platforms, a pathname is absolute when its prefix is /; On Windows platforms, a pathname is absolute when its prefix is a drive specifier followed by \ or when its prefix is .</td>
</tr>
<tr>
<td>String toString()</td>
<td>A synonym for getPath().</td>
</tr>
</tbody>
</table>

Table 11-1 refers to IOException, which is the common exception superclass for those exception classes that describe various kinds of I/O errors such as java.io.FileNotFoundException.

Listing 11-2 instantiates File with its pathname command-line argument and calls some of the File methods described in Table 11-1 to learn about this pathname.

Listing 11-2. Obtaining Abstract Pathname Information

```java
import java.io.File;
import java.io.IOException;

public class PathnameInfo
{
    public static void main(final String[] args) throws IOException
    {
        if (args.length != 1)
        {
            System.err.println("usage: java PathnameInfo pathname");
```
For example, when I specify `java PathnameInfo` (the period represents the current directory on my Windows 7 platform), I observe the following output:

```
Absolute path = C:\prj\dev\ljfad3\ch11\code\PathnameInfo\. 
Canonical path = C:\prj\dev\ljfad3\ch11\code\PathnameInfo
Name = .
Parent = null
Path = .
Is absolute = false
```

This output reveals that the canonical pathname doesn't include the period. It also shows that there is no parent pathname and that the pathname is relative.

Continuing, I now specify `java PathnameInfo c:\reports\2013\.\2012\February`. This time, I observe the following output:

```
Absolute path = c:\reports\2013\.\2012\February
Canonical path = C:\reports\2012\February
Name = February
Parent = c:\reports\2013\.\2012
Path = c:\reports\2013\.\2012\February
Is absolute = true
```

This output reveals that the canonical pathname doesn't include 2013. It also shows that the pathname is absolute.

For my final example, suppose I specify `java PathnameInfo ""` to obtain information for the empty pathname. In response, this application generates the following output:

```
Absolute path = C:\prj\dev\ljfad3\ch11\code\PathnameInfo
Canonical path = C:\prj\dev\ljfad3\ch11\code\PathnameInfo
Name =
Parent = null
Path =
Is absolute = false
```

The output reveals that `getName()` and `getPath()` return the empty string ("") because the empty pathname is empty.
Learning About a Pathname’s File or Directory

You can interrogate the filesystem to learn about the file or directory represented by a File object’s stored pathname by calling the methods that are described in Table 11-2.

Table 11-2. File Methods for Learning About a File or Directory

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean exists()</td>
<td>Returns true if and only if the file or directory denoted by this File object’s abstract pathname exists.</td>
</tr>
<tr>
<td>boolean isDirectory()</td>
<td>Returns true when this File object’s abstract pathname refers to an existing directory.</td>
</tr>
<tr>
<td>boolean isFile()</td>
<td>Returns true when this File object’s abstract pathname refers to an existing normal file. A file is normal when it’s not a directory and satisfies other platform-dependent criteria: it’s not a symbolic link or a named pipe, for example. Any nondirectory file created by a Java application is guaranteed to be a normal file.</td>
</tr>
<tr>
<td>boolean isHidden()</td>
<td>Returns true when the file denoted by this File object’s abstract pathname is hidden. The exact definition of hidden is platform dependent. On Unix/Linux platforms, a file is hidden when its name begins with a period character. On Windows platforms, a file is hidden when it has been marked as such in the filesystem.</td>
</tr>
<tr>
<td>long lastModified()</td>
<td>Returns the time that the file denoted by this File object’s abstract pathname was last modified, or 0 when the file doesn’t exist or an I/O error occurred during this method call. The returned value is measured in milliseconds since the Unix epoch (00:00:00 GMT, January 1, 1970).</td>
</tr>
<tr>
<td>long length()</td>
<td>Returns the length of the file denoted by this File object’s abstract pathname. The return value is unspecified when the pathname denotes a directory and will be 0 when the file doesn’t exist.</td>
</tr>
</tbody>
</table>

Listing 11-3 instantiates File with its pathname command-line argument, and calls all of the File methods described in Table 11-2 to learn about the pathname’s file/directory.

Listing 11-3. Obtaining File/Directory Information

```java
import java.io.File;
import java.io.IOException;
import java.util.Date;

public class FileDirectoryInfo
{
    public static void main(final String[] args) throws IOException
    {
        if (args.length != 1)
        {
            System.err.println("usage: java FileDirectoryInfo pathname");
```
File file = new File(args[0]);
System.out.println("About " + file + ":");
System.out.println("Exists = " + file.exists());
System.out.println("Is directory = " + file.isDirectory());
System.out.println("Is file = " + file.isFile());
System.out.println("Is hidden = " + file.isHidden());
System.out.println("Last modified = " + new Date(file.lastModified()));
System.out.println("Length = " + file.length());
}
}

For example, suppose I have a three-byte file named x.dat. When I specify java FileDirectoryInfo x.dat, I observe the following output:

About x.dat:
Exists = true
Is directory = false
Is file = true
Is hidden = false
Last modified = Mon Oct 14 15:31:04 CDT 2013
Length = 3

Obtaining Disk Space Information

A partition is a platform-specific portion of storage for a filesystem, for example, C:\. Obtaining the amount of partition free space is important to installers and other applications. Until Java 6 arrived, the only portable way to accomplish this task was to guess by creating files of different sizes.

Java 6 added to the File class long getFreeSpace(), long getTotalSpace(), and long getUsableSpace() methods that return space information about the partition described by the File instance’s abstract pathname. Android also supports these methods:

- long getFreeSpace() returns the number of unallocated bytes in the partition identified by this File object’s abstract pathname; it returns zero when the abstract pathname doesn’t name a partition.
- long getTotalSpace() returns the size (in bytes) of the partition identified by this File object’s abstract pathname; it returns zero when the abstract pathname doesn’t name a partition.
- long getUsableSpace() returns the number of bytes available to the current virtual machine on the partition identified by this File object’s abstract pathname; it returns zero when the abstract pathname doesn’t name a partition.

Although getFreeSpace() and getUsableSpace() appear to be equivalent, they differ in the following respect: unlike getFreeSpace(), getUsableSpace() checks for write permissions and other platform restrictions, resulting in a more accurate estimate.
CHAPTER 11: Performing Classic I/O

Note The getFreeSpace() and getUsableSpace() methods return a hint (not a guarantee) that a Java application can use all (or most) of the unallocated or available bytes. These values are a hint because a program running outside the virtual machine can allocate partition space, resulting in actual unallocated and available values being lower than the values returned by these methods.

Listing 11-4 presents an application that demonstrates these methods. After obtaining an array of all available filesystem roots, this application obtains and outputs the free, total, and usable space for each partition identified by the array.

Listing 11-4. Outputting the Free, Usable, and Total Space on All Partitions

```java
import java.io.File;

public class PartitionSpace {
    public static void main(String[] args) {
        File[] roots = File.listRoots();
        for (File root : roots) {
            System.out.println("Partition: " + root);
            System.out.println("Free space on this partition = " + root.getFreeSpace());
            System.out.println("Usable space on this partition = " + root.getUsableSpace());
            System.out.println("Total space on this partition = " + root.getTotalSpace());
            System.out.println("***");
        }
    }
}
```

Compile Listing 11-4 (javac PartitionSpace.java) and run the application (java PartitionSpace). When run on my Windows 7 machine with a hard drive designated as C:, a DVD drive designated as D:, a USB drive designated as E:, and an extra drive designated as F:, I observe the following output (usually with different free/usable space amounts on C: and E:):

Partition: C:\
Free space on this partition = 374407311360
Usable space on this partition = 374407311360
Total space on this partition = 499808989184
***
Partition: D:\
Free space on this partition = 0
Usable space on this partition = 0
Total space on this partition = 0
***
Listing Directories

File declares five methods that return the names of files and directories located in the directory identified by a File object’s abstract pathname. Table 11-3 describes these methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String[] list()</td>
<td>Returns a potentially empty array of strings naming the files and directories in the directory denoted by this File object’s abstract pathname. If the pathname doesn’t denote a directory, or if an I/O error occurs, this method returns null. Otherwise, it returns an array of strings, one string for each file or directory in the directory. Names denoting the directory itself and the directory’s parent directory are not included in the result. Each string is a filename rather than a complete path. Also, there is no guarantee that the name strings in the resulting array will appear in alphabetical or any other order.</td>
</tr>
<tr>
<td>String[] list(FilenameFilter filter)</td>
<td>A convenience method for calling list() and returning only those Strings that satisfy filter.</td>
</tr>
<tr>
<td>File[] listFiles()</td>
<td>A convenience method for calling list(), converting its array of Strings to an array of Files, and returning the Files array.</td>
</tr>
<tr>
<td>File[] listFiles(FileFilter filter)</td>
<td>A convenience method for calling list(), converting its array of Strings to an array of Files, but only for those Strings that satisfy filter, and returning the Files array.</td>
</tr>
<tr>
<td>File[] listFiles(FilenameFilter filter)</td>
<td>A convenience method for calling list(), converting its array of Strings to an array of Files, but only for those Strings that satisfy filter, and returning the Files array.</td>
</tr>
</tbody>
</table>

The overloaded list() methods return arrays of Strings denoting file and directory names. The second method lets you return only those names of interest (such as only those names that end with extension .txt) via a java.io.FilenameFilter-based filter object.
The FilenameFilter interface declares a single boolean accept(File dir, String name) method that is called for each file/directory located in the directory identified by the File object’s pathname:

- dir identifies the parent portion of the pathname (the directory path).
- name identifies the final directory name or the filename portion of the pathname.

The accept() method uses the arguments passed to these parameters to determine whether or not the file or directory satisfies its criteria for what is acceptable. It returns true when the file/directory name should be included in the returned array; otherwise, this method returns false.

Listing 11-5 presents a Dir(ectory) application that uses list(FilenameFilter) to obtain only those names that end with a specific extension.

Listing 11-5. Listing Specific Names

```java
import java.io.File;
import java.io.FilenameFilter;

public class Dir
{
    public static void main(final String[] args)
    {
        if (args.length != 2)
        {
            System.err.println("usage: java Dir dirpath ext");
            return;
        }
        File file = new File(args[0]);
        FilenameFilter fnf = new FilenameFilter()
        {
            @Override
            public boolean accept(File dir, String name)
            {
                return name.endsWith(args[1]);
            }
        };
        String[] names = file.list(fnf);
        for (String name: names)
            System.out.println(name);
    }
}
```

When I, for example, specify java Dir c:\windows.exe on my Windows 7 platform, Dir outputs only those \windows directory filenames that have the .exe extension:

```
bfsvc.exe
explorer.exe
fveupdate.exe
HelpPane.exe
hh.exe
notepad.exe
```
The overloaded listFiles() methods return arrays of Files. For the most part, they’re symmetrical with their list() counterparts. However, listFiles(FileFilter) introduces an asymmetry.

The java.io.FileFilter interface declares a single boolean accept(String pathname) method that is called for each file/directory located in the directory identified by the File object’s pathname; the argument passed to pathname identifies the complete path of the file or directory.

The accept() method uses this argument to determine whether or not the file or directory satisfies its criteria for what is acceptable. It returns true when the file/directory name should be included in the returned array; otherwise, this method returns false.

Note Because each interface’s accept() method accomplishes the same task, you might be wondering which interface to use. If you prefer a path broken into its directory and name components, use FilenameFilter. However, if you prefer a complete pathname, use FileFilter; you can always call getParent() and getName() to get these components.

Creating and Manipulating Files and Directories

File also declares several methods for creating new files and directories and manipulating existing files and directories. Table 11-4 describes these methods.

Table 11-4. File Methods for Creating New and Manipulating Existing Files and Directories

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean createNewFile()</td>
<td>Atomically creates a new, empty file named by this File object’s abstract pathname if and only if a file with this name doesn’t yet exist. The check for file existence (and the creation of the file when it doesn’t exist) is a single operation that’s atomic with respect to all other filesystem activities that might affect the file. This method returns true when the named file doesn’t exist and was successfully created, and returns false when the named file already exists. It throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>static File createTempFile(String prefix, String suffix)</td>
<td>Creates an empty file in the default temporary file directory using the given prefix and suffix to generate its name. This overloaded class method calls its three-parameter variant, passing prefix, suffix, and null to this other method, and returning the other method’s return value.</td>
</tr>
</tbody>
</table>
### Table 11-4. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static File createTempFile(String prefix,</td>
<td>Creates an empty file in the specified directory using the given prefix and suffix to generate its name. The name begins with the character sequence specified by prefix and ends with the character sequence specified by suffix; .tmp is used as the suffix when suffix is null. This method returns the created file's pathname when successful. It throws java.lang.IllegalArgumentException when prefix contains fewer than three characters and IOException when the file couldn't be created.</td>
</tr>
<tr>
<td>String suffix, File directory)</td>
<td></td>
</tr>
<tr>
<td>boolean delete()</td>
<td>Deletes the file or directory denoted by this File object's pathname. Returns true when successful; otherwise, returns false. If the pathname denotes a directory, the directory must be empty in order to be deleted.</td>
</tr>
<tr>
<td>void deleteOnExit()</td>
<td>Requests that the file or directory denoted by this File object's abstract pathname be deleted when the virtual machine terminates. Reinvoke this method on the same File object has no effect. Once deletion has been requested, it's not possible to cancel the request. Therefore, this method should be used with care.</td>
</tr>
<tr>
<td>boolean mkdir()</td>
<td>Creates the directory named by this File object's abstract pathname. Returns true when successful; otherwise, returns false.</td>
</tr>
<tr>
<td>boolean mkdirs()</td>
<td>Creates the directory and any necessary intermediate directories named by this File object's abstract pathname. Returns true when successful; otherwise, returns false.</td>
</tr>
<tr>
<td>boolean renameTo(File dest)</td>
<td>Renames the file denoted by this File object's abstract pathname to dest. Returns true when successful; otherwise, returns false. This method throws NullPointerException when dest is null.</td>
</tr>
<tr>
<td>boolean setLastModified(long time)</td>
<td>Sets the last-modified time of the file or directory named by this File object's abstract pathname. Returns true when successful; otherwise, returns false. This method throws IllegalArgumentException when time is negative.</td>
</tr>
<tr>
<td></td>
<td>All platforms support file-modification times to the nearest second, but some provide more precision. The time value will be truncated to fit the supported precision. If the operation succeeds and no intervening operations on the file take place, the next call to lastModified() will return the (possibly truncated) time value passed to this method.</td>
</tr>
</tbody>
</table>
Suppose you’re designing a text-editor application that a user will use to open a text file and make changes to its content. Until the user explicitly saves these changes to the file, you want the text file to remain unchanged.

Because the user doesn’t want to lose these changes when the application crashes or the computer loses power, you design the application to save these changes to a temporary file every few minutes. This way, the user has a backup of the changes.

You can use the overloaded `createTempFile()` methods to create the temporary file. If you don’t specify a directory in which to store this file, it’s created in the directory identified by the `java.io.tmpdir` system property.

You probably want to remove the temporary file after the user tells the application to save or discard the changes. The `deleteOnExit()` method lets you register a temporary file for deletion; it’s deleted when the virtual machine ends without a crash/power loss.

Listing 11-6 presents a `TempFileDemo` application for experimenting with the `createTempFile()` and `deleteOnExit()` methods.

Listing 11-6. Experimenting with Temporary Files

```java
import java.io.File;
import java.io.IOException;

public class TempFileDemo
{
    public static void main(String[] args) throws IOException
    {
        System.out.println(System.getProperty("java.io.tmpdir"));
        File temp = File.createTempFile("text", ".txt");
        System.out.println(temp);
        temp.deleteOnExit();
    }
}
```

After outputting the location where temporary files are stored, `TempFileDemo` creates a temporary file whose name begins with `text` and ends with the `.txt` extension. `TempFileDemo` next outputs the temporary file’s name and registers the temporary file for deletion upon the successful termination of the application.

I observed the following output during one run of `TempFileDemo` (and the file disappeared on exit):

```
C:\Users\Owner\AppData\Local\Temp\text3173127870811188221.txt
```

### Setting and Getting Permissions

Java 1.2 added a boolean `setReadOnly()` method to the `File` class to mark a file or directory as read-only. However, a method to revert the file or directory to the writable state wasn’t added. More importantly, until Java 6’s arrival, `File` offered no way to manage an abstract pathname’s read, write, and execute permissions.
Java 6 added to File boolean setExecutable(boolean executable), boolean setExecutable(boolean executable, boolean ownerOnly), boolean setReadable(boolean readable), boolean setReadable(boolean readable, boolean ownerOnly), boolean setWritable(boolean writable), and boolean setWritable(boolean writable, boolean ownerOnly) methods that let you set the owner's or everybody's execute, read, and write permissions for the file identified by the File object's abstract pathname. Android also supports these methods:

- **boolean setExecutable(boolean executable, boolean ownerOnly)** enables (pass true to executable) or disables (pass false to executable) this abstract pathname's execute permission for its owner (pass true to ownerOnly) or everyone (pass false to ownerOnly). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions or when executable is false and the filesystem doesn't implement an execute permission.

- **boolean setExecutable(boolean executable)** is a convenience method that invokes the previous method to set the execute permission for the owner.

- **boolean setReadable(boolean readable, boolean ownerOnly)** enables (pass true to readable) or disables (pass false to readable) this abstract pathname's read permission for its owner (pass true to ownerOnly) or everyone (pass false to ownerOnly). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions or when readable is false and the filesystem doesn't implement a read permission.

- **boolean setReadable(boolean readable)** is a convenience method that invokes the previous method to set the read permission for the owner.

- **boolean setWritable(boolean writable, boolean ownerOnly)** enables (pass true to writable) or disables (pass false to writable) this abstract pathname's write permission for its owner (pass true to ownerOnly) or everyone (pass false to ownerOnly). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions.

- **boolean setWritable(boolean writable)** is a convenience method that invokes the previous method to set the write permission for the owner.

Along with these methods, Java 6 retrofitted File's boolean canRead() and boolean canWrite() methods, and introduced a boolean canExecute() method to return an abstract pathname's access permissions. These methods return true when the file or directory object identified by the abstract pathname exists and when the appropriate permission is in effect. For example, canWrite() returns true when the abstract pathname exists and when the application has permission to write to the file.

The canRead(), canWrite(), and canExecute() methods can be used to implement a simple utility that identifies which permissions have been assigned to an arbitrary file or directory. This utility's source code is presented in Listing 11-7.
Listing 11-7. Checking a File’s or Directory’s Permissions

```java
import java.io.File;

public class Permissions
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java Permissions filespec");
            return;
        }
        File file = new File(args[0]);
        System.out.println("Checking permissions for " + args[0]);
        System.out.println("  Execute = " + file.canExecute());
        System.out.println("  Read = " + file.canRead());
        System.out.println("  Write = " + file.canWrite());
    }
}
```

Compile Listing 11-7 (javac Permissions.java). Assuming a readable and executable (only) file named x in the current directory, `java Permissions x` generates the following output:

Checking permissions for x
  Execute = true
  Read = true
  Write = false

Exploring Miscellaneous Capabilities

Finally, File implements the java.lang.Comparable interface’s compareTo() method and overrides equals() and hashCode(). Table 11-5 describes these miscellaneous methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int compareTo(File pathname)</td>
<td>Compares two pathnames lexicographically. The ordering defined by this method depends upon the underlying platform. On Unix/Linux platforms, alphabetic case is significant when comparing pathnames; on Windows platforms, alphabetic case is insignificant. Returns zero when pathname’s abstract pathname equals this File object’s abstract pathname, a negative value when this File object’s abstract pathname is less than pathname, and a positive value otherwise. To accurately compare two File objects, call getCanonicalFile() on each File object and then compare the returned File objects.</td>
</tr>
</tbody>
</table>

(continued)
Table 11-5. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object obj)</td>
<td>Compares this File object with obj for equality. Abstract pathname equality depends upon the underlying platform. On Unix/Linux platforms, alphabetic case is significant when comparing pathnames; on Windows platforms, alphabetic case is insignificant. Returns true if and only if obj is not null and is a File object whose abstract pathname denotes the same file/directory as this File object’s abstract pathname.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Calculates and returns a hash code for this pathname. This calculation depends upon the underlying platform. On Unix/Linux platforms, a pathname’s hash code equals the exclusive OR of its pathname string’s hash code and decimal value 1234321. On Windows platforms, the hash code is the exclusive OR of the lowercased pathname string’s hash code and decimal value 1234321. The current locale (geographical, political, or cultural region) is not taken into account when lowercasing the pathname string.</td>
</tr>
</tbody>
</table>

Listing 11-8 presents an application that demonstrates `compareTo()` along with `getCanonicalFile()`.

Listing 11-8. Comparing Files

```java
import java.io.File;
import java.io.IOException;

public class Compare {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.err.println("usage: java Compare filespec1 filespec2");
            return;
        }

        File file1 = new File(args[0]);
        File file2 = new File(args[1]);
        System.out.println(file1.compareTo(file2));
        System.out.println(file1.getCanonicalFile().compareTo(file2.getCanonicalFile()));
    }
}
```
Compile Listing 11-8 (javac Compare.java). Assuming successful compilation and a Windows platform, execute the following command line:

```java
java Compare Compare.class \Compare.class
```

You should observe the following output:

```
53
0
```

The 53 indicates that file1’s abstract pathname is lexicographically greater than file2’s abstract pathname. However, when comparing their canonical representations, these abstract pathnames are considered to be identical (as indicated by the 0).

## Working with the RandomAccessFile API

Files can be created and/or opened for **random access** in which a mixture of write and read operations can occur until the file is closed. Java supports this random access via its concrete `java.io.RandomAccessFile` class.

**Note**  RandomAccessFile has its place in Android app development. For example, you can use this class to read an app’s raw resource file. To learn how, check out “RandomAccessFile in Android raw resource file” ([http://stackoverflow.com/questions/9335379/randomaccessfile-in-android-raw-resource-file](http://stackoverflow.com/questions/9335379/randomaccessfile-in-android-raw-resource-file)).

RandomAccessFile declares the following constructors:

- `RandomAccessFile(File file, String mode)` creates and opens a new file if it doesn’t exist or opens an existing file. The file is identified by `file`’s abstract pathname and is created and/or opened according to `mode`.

- `RandomAccessFile(String pathname, String mode)` creates and opens a new file if it doesn’t exist or opens an existing file. The file is identified by `pathname` and is created and/or opened according to `mode`.

Either constructor’s `mode` argument must be one of "r", "rw", "rws", or "rwd"; otherwise, the constructor throws `IllegalArgumentException`. These string literals have the following meanings:

- "r" informs the constructor to open an existing file for reading only. Any attempt to write to the file results in a thrown instance of the `IOException` class.

- "rw" informs the constructor to create and open a new file when it doesn’t exist for reading and writing or open an existing file for reading and writing.
"rwd" informs the constructor to create and open a new file when it doesn’t exist for reading and writing or open an existing file for reading and writing. Furthermore, each update to the file’s content must be written synchronously to the underlying storage device.

"rws" informs the constructor to create and open a new file when it doesn’t exist for reading and writing or open an existing file for reading and writing. Furthermore, each update to the file’s content or metadata must be written synchronously to the underlying storage device.

**Note** A file’s *metadata* is data about the file and not actual file contents. Examples of metadata include the file’s length and the time the file was last modified.

The "rwd" and "rws" modes ensure than any writes to a file located on a local storage device are written to the device, which guarantees that critical data isn’t lost when the operating system crashes. No guarantee is made when the file doesn’t reside on a local device.

**Note** Operations on a random access file opened in "rwd" or "rws" mode are slower than these same operations on a random access file opened in "rw" mode.

These constructors throw FileNotFoundException when mode is "r" and the file identified by pathname cannot be opened (it might not exist or it might be a directory) or when mode is "rw" and pathname is read-only or a directory.

The following example demonstrates the second constructor by attempting to open an existing file for read access via the "r" mode string:

```java
RandomAccessFile raf = new RandomAccessFile("employee.dat", "r");
```

A random access file is associated with a *file pointer*, a cursor that identifies the location of the next byte to write or read. When an existing file is opened, the file pointer is set to its first byte at offset 0. The file pointer is also set to 0 when the file is created.

Write or read operations start at the file pointer and advance it past the number of bytes written or read. Operations that write past the current end of the file cause the file to be extended. These operations continue until the file is closed.

RandomAccessFile declares a wide variety of methods. I present a representative sample of these methods in Table 11-6.
Table 11-6: *RandomAccessFile* Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
<td>Closes the file and releases any associated platform resources. Subsequent writes or reads result in <code>IOException</code>. Also, the file cannot be reopened with this <code>RandomAccessFile</code> object. This method throws <code>IOException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>FileDescriptor getFD()</td>
<td>Returns the file's associated file descriptor object. This method throws <code>IOException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>long getFilePointer()</td>
<td>Returns the file pointer's current zero-based byte offset into the file. This method throws <code>IOException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>long length()</td>
<td>Returns the length (measured in bytes) of the file. This method throws <code>IOException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>int read()</td>
<td>Reads and returns (as an int in the range 0 to 255) the next byte from the file or returns -1 when the end of the file is reached. This method blocks when no input is available and throws <code>IOException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>int read(byte[] b)</td>
<td>Reads up to <code>b.length</code> bytes of data from the file into byte array <code>b</code>. This method blocks until at least 1 byte of input is available. It returns the number of bytes read into the array, or returns -1 when the end of the file is reached. It throws <code>NullPointerException</code> when <code>b</code> is null and <code>IOException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>char readChar()</td>
<td>Reads and returns a character from the file. This method reads 2 bytes from the file starting at the current file pointer. If the bytes read, in order, are <code>b1</code> and <code>b2</code>, where <code>0 &lt;= b1, b2 &lt;= 255</code>, the result is equal to (char) `((b1 &lt;&lt; 8)</td>
</tr>
<tr>
<td>int readInt()</td>
<td>Reads and returns a 32-bit integer from the file. This method reads 4 bytes from the file starting at the current file pointer. If the bytes read, in order, are <code>b1, b2, b3, and b4</code>, where <code>0 &lt;= b1, b2, b3, b4 &lt;= 255</code>, the result is equal to `((b1 &lt;&lt; 24)</td>
</tr>
<tr>
<td>void seek(long pos)</td>
<td>Sets the file pointer's current offset to <code>pos</code> (which is measured in bytes from the beginning of the file). If the offset is set beyond the end of the file, the file's length doesn't change. The file length will only change by writing after the offset has been set beyond the end of the file. This method throws <code>IOException</code> when the value in <code>pos</code> is negative or when an I/O error occurs.</td>
</tr>
</tbody>
</table>
Table 11-6. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setLength(long newLength)</td>
<td>Sets the file's length. If the present length as returned by length() is greater than newLength, the file is truncated. In this case, if the file offset as returned by getFilePointer() is greater than newLength, the offset will be equal to newLength after setLength() returns. If the present length is smaller than newLength, the file is extended. In this case, the contents of the extended portion of the file are not defined. This method throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>int skipBytes(int n)</td>
<td>Attempts to skip over n bytes. This method skips over a smaller number of bytes (possibly zero) when the end of file is reached before n bytes have been skipped. It doesn't throw EOFException in this situation. If n is negative, no bytes are skipped. The actual number of bytes skipped is returned. This method throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>void write(byte[] b)</td>
<td>Writes b.length bytes from byte array b to the file starting at the current file pointer position. This method throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>void write(int b)</td>
<td>Writes the lower 8 bits of b to the file at the current file pointer position. This method throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>void writeChars(String s)</td>
<td>Writes string s to the file as a sequence of characters starting at the current file pointer position. This method throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>void writeInt(int i)</td>
<td>Writes 32-bit integer i to the file starting at the current file pointer position. The 4 bytes are written with the high byte first. This method throws IOException when an I/O error occurs.</td>
</tr>
</tbody>
</table>

Most of Table 11-6's methods are fairly self-explanatory. However, the getFD() method requires further enlightenment.

**Note** RandomAccessFile's read-prefixed methods and skipBytes() originate in the java.io.DataInput interface, which this class implements. Furthermore, RandomAccessFile's write-prefixed methods originate in the java.io.DataOutput interface, which this class also implements.

When a file is opened, the underlying platform creates a platform-dependent structure to represent the file. A handle to this structure is stored in an instance of the java.io.FileDescriptor class, which getFD() returns.

**Note** A handle is an identifier that Java passes to the underlying platform to identify, in this case, a specific open file when it requires that the underlying platform perform a file operation.
FileDescriptor is a small class that declares three FileDescriptor constants named in, out, and err. These constants let System.in, System.out, and System.err (discussed later in this chapter) provide access to the standard input, standard output, and standard error streams.

FileDescriptor also declares the following pair of methods:

- `void sync()` tells the underlying platform to flush (empty) the contents of the open file's output buffers to their associated local disk device. `sync()` returns after all modified data and attributes have been written to the relevant device. It throws java.io.SyncFailedException when the buffers cannot be flushed or because the platform cannot guarantee that all the buffers have been synchronized with physical media.

- `boolean valid()` determines whether or not this file descriptor object is valid. It returns true when the file descriptor object represents an open file or other active I/O connection; otherwise, it returns false.

Data that is written to an open file ends up being stored in the underlying platform's output buffers. When the buffers fill to capacity, the platform empties them to the disk. Buffers improve performance because disk access is slow.

However, when you write data to a random access file that's been opened via mode "rwd" or "rws", each write operation's data is written straight to the disk. As a result, write operations are slower than when the random access file is opened in "rw" mode.

Suppose you have a situation that combines writing data through the output buffers and writing data directly to the disk. The following example addresses this hybrid scenario by opening the file in mode "rw" and selectively calling FileDescriptor's sync() method.

```java
RandomAccessFile raf = new RandomAccessFile("employee.dat", "rw");
FileDescriptor fd = raf.getFD();
// Perform a critical write operation.
raf.write(...);
// Synchronize with underlying disk by flushing platform's output buffers to disk.
fd.sync();
// Perform non-critical write operation where synchronization isn't necessary.
raf.write(...);
// Do other work.
// Close file, emptying output buffers to disk.
raf.close();
```

RandomAccessFile is useful for creating a flat file database, a single file organized into records and fields. A record stores a single entry (such as a part in a parts database) and a field stores a single attribute of the entry (such as a part number).

---

**Note** The term field is also used to refer to a variable declared within a class. To avoid confusion with this overloaded terminology, think of a field variable as being analogous to a record's field attribute.
A flat file database typically organizes its content into a sequence of fixed-length records. Each record is further organized into one or more fixed-length fields. Figure 11-1 illustrates this concept in the context of a parts database.

![Diagram of a flat file database with records and fields]

According to Figure 11-1, each field has a name (partnum, desc, qty, and ucost). Also, each record is assigned a number starting at 0. This example consists of five records, of which only three are shown for brevity.

To show you how to implement a flat file database in terms of RandomAccessFile, I've created a simple PartsDB class to model Figure 11-1. Check out Listing 11-9.

Listing 11-9. Implementing the Parts Flat File Database

```java
import java.io.IOException;
import java.io.RandomAccessFile;

public class PartsDB {
    public final static int PNUMLEN = 20;
    public final static int DESCLEN = 30;
    public final static int QUANLEN = 4;
    public final static int COSTLEN = 4;
    private final static int RECLEN = 2 * PNUMLEN + 2 * DESCLEN + QUANLEN + COSTLEN;

    private RandomAccessFile raf;

    public PartsDB(String pathname) throws IOException {
        raf = new RandomAccessFile(pathname, "rw");
    }

    public void append(String partnum, String partdesc, int qty, int ucost) throws IOException {
        raf.seek(raf.length());
        write(partnum, partdesc, qty, ucost);
    }

    private void write(String partnum, String partdesc, int qty, int ucost) throws IOException {
        // Write to file
    }
}
```

Figure 11-1. A flat file database of automotive parts is divided into records and fields

*Figure 11-1 shows a portion of a flat file database database with fields: partnum, desc, qty, and ucost.*
public void close()
{
    try {
        raf.close();
    } catch (IOException ioe) {
        System.err.println(ioe);
    }
}

public int numRecs() throws IOException {
    return (int) raf.length() / RECLEN;
}

public Part select(int recno) throws IOException {
    if (recno < 0 || recno >= numRecs())
        throw new IllegalArgumentException(recno + " out of range");
    raf.seek(recno * RECLEN);
    return read();
}

public void update(int recno, String partnum, String partdesc, int qty, int ucost) throws IOException {
    if (recno < 0 || recno >= numRecs())
        throw new IllegalArgumentException(recno + " out of range");
    raf.seek(recno * RECLEN);
    write(partnum, partdesc, qty, ucost);
}

private Part read() throws IOException {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < PNUMLEN; i++)
        sb.append(raf.readChar());
    String partnum = sb.toString().trim();
    sb.setLength(0);
    for (int i = 0; i < DESCLEN; i++)
        sb.append(raf.readChar());
    String partdesc = sb.toString().trim();
    int qty = raf.readInt();
    int ucost = raf.readInt();
    return new Part(partnum, partdesc, qty, ucost);
}
private void write(String partnum, String partdesc, int qty, int ucost)
    throws IOException
{
    StringBuffer sb = new StringBuffer(partnum);
    if (sb.length() > PNUMLEN)
        sb.setLength(PNUMLEN);
    else
    if (sb.length() < PNUMLEN)
    {
        int len = PNUMLEN - sb.length();
        for (int i = 0; i < len; i++)
            sb.append(" ");
    }
    raf.writeChars(sb.toString());
    sb = new StringBuffer(partdesc);
    if (sb.length() > DESCLEN)
        sb.setLength(DESCLEN);
    else
    if (sb.length() < DESCLEN)
    {
        int len = DESCLEN - sb.length();
        for (int i = 0; i < len; i++)
            sb.append(" ");
    }
    raf.writeChars(sb.toString());
    raf.writeInt(qty);
    raf.writeInt(ucost);
}

public static class Part
{
    private String partnum;
    private String desc;
    private int qty;
    private int ucost;

    public Part(String partnum, String desc, int qty, int ucost)
    {
        this.partnum = partnum;
        this.desc = desc;
        this.qty = qty;
        this.ucost = ucost;
    }

    String getDesc()
    {
        return desc;
    }

    String getPartnum()
    {
        return partnum;
    }
}
int getQty()
{
    return qty;
}

int getUnitCost()
{
    return ucost;
}
}

PartsDB first declares constants that identify the lengths of the string and 32-bit integer fields. It then declares a constant that calculates the record length in terms of bytes. The calculation takes into account the fact that a character occupies 2 bytes in the file.

These constants are followed by a field named raf that is of type RandomAccessFile. This field is assigned an instance of the RandomAccessFile class in the subsequent constructor, which creates/opens a new file or opens an existing file because of "rw".

PartsDB next declares append(), close(), numRecs(), select(), and update(). These methods append a record to the file, close the file, return the number of records in the file, select and return a specific record, and update a specific record.

- The append() method first calls length() and seek(). Doing so ensures that the file pointer is positioned to the end of the file before calling the private write() method to write a record containing this method's arguments.

- RandomAccessFile's close() method can throw IOException. Because this is a rare occurrence, I chose to handle this exception in PartDB's close() method, which keeps that method's signature simple. However, I print a message when IOException occurs.

- The numRecs() method returns the number of records in the file. These records are numbered starting with 0 and ending with numRecs() - 1. Each of the select() and update() methods verifies that its recno argument lies within this range.

- The select() method calls the private read() method to return the record identified by recno as an instance of the nested Part class. Part's constructor initializes a Part object to a record's field values, and its getter methods return these values.

- The update() method is equally simple. As with select(), it first positions the file pointer to the start of the record identified by recno. As with append(), it calls write() to write out its arguments but replaces a record instead of adding one.

Records are written with the private write() method. Because fields must have exact sizes, write() pads String-based values that are shorter than a field size with spaces on the right and truncates these values to the field size when needed.

Records are read via the private read() method. read() removes the padding before saving a String-based field value in the Part object.
By itself, PartsDB is useless. You need an application that lets you experiment with this class, and Listing 11-10 fulfills this requirement.

**Listing 11-10. Experimenting with the Parts Flat File Database**

```java
import java.io.IOException;

public class UsePartsDB {
    public static void main(String[] args) {
        PartsDB pdb = null;
        try {
            pdb = new PartsDB("parts.db");
            if (pdb.numRecs() == 0) {
                // Populate the database with records.
                pdb.append("1-9009-3323-4x", "Wiper Blade Micro Edge", 30, 2468);
                pdb.append("1-3233-44923-7j", "Parking Brake Cable", 5, 1439);
                pdb.append("2-3399-6693-2m", "Halogen Bulb H4 55/60W", 22, 813);
                pdb.append("2-599-2029-6k", "Turbo Oil Line O-Ring ", 26, 155);
                pdb.append("3-1299-3299-9u", "Air Pump Electric", 9, 20200);
            }
            dumpRecords(pdb);
            pdb.update(1, "1-3233-44923-7j", "Parking Brake Cable", 5, 1995);
            dumpRecords(pdb);
        } catch (IOException ioe) {
            System.err.println(ioe);
        } finally {
            if (pdb != null) pdb.close();
        }
    }

    static void dumpRecords(PartsDB pdb) throws IOException {
        for (int i = 0; i < pdb.numRecs(); i++) {
            PartsDB.Part part = pdb.select(i);
            System.out.print(format(part.getPartnum(), PartsDB.PNUMLEN, true));
            System.out.print(" | ");
            System.out.print(format(part.getDesc(), PartsDB.DESCLEN, true));
            System.out.print(" | ");
            System.out.print(format(part.getUnitCost() / 100 + "." + part.getUnitCost() % 100, 10, false));
            String s = part.getUnitCost() / 100 + "." + part.getUnitCost() % 100;
            if (s.charAt(s.length() - 2) == '.') s += "0";
        }
    }
}
```
System.out.println(format(s, 10, false));
}
System.out.println("Number of records = " + pdb.numRecs());
System.out.println();
}

static String format(String value, int maxWidth, boolean leftAlign)
{
    StringBuffer sb = new StringBuffer();
    int len = value.length();
    if (len > maxWidth)
    {
        len = maxWidth;
        value = value.substring(0, len);
    }
    if (leftAlign)
    {
        sb.append(value);
        for (int i = 0; i < maxWidth-len; i++)
            sb.append( " ");
    }
    else
    {
        for (int i = 0; i < maxWidth-len; i++)
            sb.append( " ");
        sb.append(value);
    }
    return sb.toString();
}

Listing 11-10's main() method begins by instantiating PartsDB, with parts.db as the name of the database file. When this file has no records, numRecs() returns 0 and several records are appended to the file via the append() method.

main() next dumps the five records stored in parts.db to the standard output stream, updates the unit cost in the record whose number is 1, once again dumps these records to the standard output stream to show this change, and closes the database.

Note: I store unit cost values as integer-based penny amounts. For example, I specify literal 1995 to represent 1995 pennies, or $19.95. If I were to use java.math.BigDecimal objects to store currency values, I would have to refactor PartsDB to take advantage of object serialization, and I'm not prepared to do that right now. (I discuss object serialization later in this chapter.)

main() relies on a dumpRecords() helper method to dump these records, and dumpRecords() relies on a format() helper method to format field values so that they can be presented in properly aligned columns—I could have used java.util.Formatter (see Chapter 13) instead. The following output reveals this alignment:
And there you have it: a simple flat file database. Despite its lack of support for advanced database features such as indexes and transaction management, a flat file database might be all that your Android application requires.

Note  To learn more about flat file databases, check out Wikipedia’s “Flat file database” entry (http://en.wikipedia.org/wiki/Flat_file_database).

**Working with Streams**

Along with File and RandomAccessFile, Java uses streams to perform I/O operations. A stream is an ordered sequence of bytes of arbitrary length. Bytes flow over an output stream from an application to a destination and flow over an input stream from a source to an application. Figure 11-2 illustrates these flows.

![Diagram of input and output streams](image)

*Figure 11-2. Conceptualizing output and input streams as flows of bytes*

Note  Java’s use of the word stream is analogous to stream of water, stream of electrons, and so on.
Java recognizes various stream destinations, such as byte arrays, files, screens, sockets (network endpoints), and thread pipes. Java also recognizes various stream sources. Examples include byte arrays, files, keyboards, sockets, and thread pipes. (I will discuss sockets in Chapter 12.)

**Stream Classes Overview**

The java.io package provides several output stream and input stream classes that are descendants of the abstract OutputStream and InputStream classes. Figure 11-3 reveals the hierarchy of output stream classes.

![InputStream hierarchy](image_url)

*Figure 11-3. All output stream classes except for PrintStream are denoted by their OutputStream suffixes*

Figure 11-4 reveals the hierarchy of input stream classes.

![InputStream hierarchy](image_url)

*Figure 11-4. Note that LineNumberInputStream and StringBufferInputStream are deprecated*
LineNumberInputStream and StringBufferInputStream have been deprecated because they don’t support different character encodings, a topic I discuss later in this chapter. LineNumberReader and StringReader are their replacements. (I discuss readers later in this chapter.)

**Note** PrintStream is another class that should be deprecated because it doesn’t support different character encodings; PrintWriter is its replacement. However, it’s doubtful that Oracle (and Google) will deprecate this class because PrintStream is the type of the System class’s out and err class fields, and too much legacy code depends upon this fact.

Other Java packages provide additional output stream and input stream classes. For example, java.util.zip provides four output stream classes that compress uncompressed data into various formats and four matching input stream classes that uncompress compressed data from the same formats:

- CheckedOutputStream
- CheckedInputStream
- DeflaterOutputStream
- GZIPOutputStream
- GZIPInputStream
-InflaterInputStream
- ZipOutputStream
- ZipInputStream

Also, the java.util.jar package provides a pair of stream classes for writing content to a JAR file and for reading content from a JAR file:

- JarOutputStream
- JarInputStream

In the next several sections, I take you on a tour of most of java.io’s output stream and input stream classes, beginning with OutputStream and InputStream.

**OutputStream and InputStream**

Java provides the OutputStream and InputStream classes for performing stream I/O. OutputStream is the superclass of all output stream subclasses. Table 11-7 describes OutputStream’s methods.
The flush() method is useful in a long-running application where you need to save changes every so often, as in the previously mentioned text-editor application that saves changes to a temporary file every few minutes. Remember that flush() only flushes bytes to the platform; doing so doesn’t necessarily result in the platform flushing these bytes to the disk.

The flush() method is useful in a long-running application where you need to save changes every so often, as in the previously mentioned text-editor application that saves changes to a temporary file every few minutes. Remember that flush() only flushes bytes to the platform; doing so doesn’t necessarily result in the platform flushing these bytes to the disk.

Note The close() method automatically flushes the output stream. If an application ends before close() is called, the output stream is automatically closed and its data is flushed.

InputStream is the superclass of all input stream subclasses. Table 11-8 describes InputStream’s methods.
Table 11-8. InputStream Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int available()</td>
<td>Returns an estimate of the number of bytes that can be read from this input stream via the next read() method call (or skipped over via skip()) without blocking the calling thread. This method throws IOException when an I/O error occurs. It’s never correct to use this method’s return value to allocate a buffer for holding all of the stream's data because a subclass might not return the total size of the stream.</td>
</tr>
<tr>
<td>void close()</td>
<td>Closes this input stream and releases any platform resources associated with the stream. This method throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>void mark(int readlimit)</td>
<td>Marks the current position in this input stream. A subsequent call to reset() repositions this stream to the last marked position so that subsequent read operations re-read the same bytes. The readlimit argument tells this input stream to allow that many bytes to be read before invalidating this mark (so that the stream cannot be reset to the marked position).</td>
</tr>
<tr>
<td>boolean markSupported()</td>
<td>Returns true when this input stream supports mark() and reset(); otherwise, returns false.</td>
</tr>
<tr>
<td>int read()</td>
<td>Reads and returns (as an int in the range 0 to 255) the next byte from this input stream, or returns -1 when the end of the stream is reached. This method blocks until input is available, the end of the stream is detected, or an exception is thrown. It throws IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>int read(byte[] b)</td>
<td>Reads some number of bytes from this input stream and stores them in byte array b. Returns the number of bytes actually read (which might be less than b’s length but is never more than this length), or returns -1 when the end of the stream is reached (no byte is available to read). This method blocks until input is available, the end of the stream is detected, or an exception is thrown. It throws NullPointerException when b is null and IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>int read(byte[] b, int off, int len)</td>
<td>Reads no more than len bytes from this input stream and stores them in byte array b, starting at the offset specified by off. Returns the number of bytes actually read (which might be less than len but is never more than len), or returns -1 when the end of the stream is reached (no byte is available to read). This method blocks until input is available, the end of the stream is detected, or an exception is thrown. It throws NullPointerException when b is null; IndexOutOfBoundsException when off is negative, len is negative, or len is greater than b.length - off; and IOException when an I/O error occurs.</td>
</tr>
<tr>
<td>void reset()</td>
<td>Repositions this input stream to the position at the time mark() was last called. This method throws IOException when this input stream has not been marked or the mark has been invalidated.</td>
</tr>
<tr>
<td>long skip(long n)</td>
<td>Skips over and discards n bytes of data from this input stream. This method might skip over some smaller number of bytes (possibly zero), for example, when the end of the file is reached before n bytes have been skipped. The actual number of bytes skipped is returned. When n is negative, no bytes are skipped. This method throws IOException when this input stream doesn’t support skipping or when some other I/O error occurs.</td>
</tr>
</tbody>
</table>
InputStream subclasses such as ByteArrayInputStream support marking the current read position in the input stream via the mark() method and later return to that position via the reset() method.

**Caution** Don’t forget to call markSupported() to find out if the subclass supports mark() and reset().

**ByteArrayOutputStream and ByteArrayInputStream**

Byte arrays are often useful as stream destinations and sources. The ByteArrayOutputStream class lets you write a stream of bytes to a byte array; the ByteArrayInputStream class lets you read a stream of bytes from a byte array.

ByteArrayOutputStream declares two constructors. Each constructor creates a byte array output stream with an internal byte array; a copy of this array can be returned by calling ByteArrayOutputStream’s byte[] toByteArray() method.

- ByteArrayOutputStream() creates a byte array output stream with an internal byte array whose initial size is 32 bytes. This array grows as necessary.
- ByteArrayOutputStream(int size) creates a byte array output stream with an internal byte array whose initial size is specified by size and grows as necessary. This constructor throws IllegalArgumentException when size is less than zero.

The following example uses ByteArrayOutputStream() to create a byte array output stream with an internal byte array set to the default size:

```java
ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

ByteArrayInputStream also declares a pair of constructors. Each constructor creates a byte array input stream based on the specified byte array and also keeps track of the next byte to read from the array and the number of bytes to read.

- ByteArrayInputStream(byte[] ba) creates a byte array input stream that uses ba as its byte array (ba is used directly; a copy isn’t created). The position is set to 0 and the number of bytes to read is set to ba.length.
- ByteArrayInputStream(byte[] ba, int offset, int count) creates a byte array input stream that uses ba as its byte array (no copy is made). The position is set to offset and the number of bytes to read is set to count.

The following example uses ByteArrayInputStream(byte[]) to create a byte array input stream whose source is a copy of the previous byte array output stream’s byte array:

```java
ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
```

ByteArrayOutputStream and ByteArrayInputStream are useful in a scenario where you need to convert an image to an array of bytes, process these bytes in some manner, and convert the bytes back to the image.
For example, suppose you're writing an Android-based image-processing application. You decode a file containing the image into an Android-specific android.graphics.Bitmap instance, compress this instance into a ByteArrayOutputStream instance, obtain a copy of the byte array output stream's array, process this array in some manner, convert this array to a ByteArrayInputStream instance, and use the byte array input stream to decode these bytes into another Bitmap instance, as follows:

```java
String pathname = "..."; // Assume a legitimate pathname to an image.
Bitmap bm = BitmapFactory.decodeFile(pathname);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
if (bm.compress(Bitmap.CompressFormat.PNG, 100, baos))
{
    byte[] imageBytes = baos.toByteArray();
    // Do something with imageBytes.
    bm = BitmapFactory.decodeStream(new ByteArrayInputStream(imageBytes));
}
```

This example obtains an image file's pathname and then calls the concrete android.graphics.BitmapFactory class's Bitmap decodeFile(String pathname) class method. This method decodes the image file identified by pathname into a bitmap and returns a Bitmap instance that represents this bitmap.

After creating a ByteArrayOutputStream object, the example uses the returned Bitmap instance to call Bitmap's boolean compress(Bitmap.CompressFormat format, int quality, OutputStream stream) method to write a compressed version of the bitmap to the byte array output stream:

- format identifies the format of the compressed image. I've chosen to use the popular Portable Network Graphics (PNG) format.
- quality hints to the compressor as to how much compression is required. This value ranges from 0 through 100, where 0 means maximum compression at the expense of quality and 100 means maximum quality at the expense of compression. Formats such as PNG ignore quality because they employ lossless compression.
- stream identifies the stream on which to write the compressed image data.

When compress() returns true, which means that it successfully compressed the image onto the byte array output stream in the PNG format, the ByteArrayOutputStream object's toByteArray() method is called to create and return a byte array with the image's bytes.

Continuing, the array is processed, a ByteArrayInputStream object is created with the processed bytes serving as the source of this stream, and BitmapFactory's Bitmap decodeStream(InputStream is) class method is called to convert the byte array input stream's source of bytes to a Bitmap instance.

### FileOutputStream and FileInputStream

Files are common stream destinations and sources. The concrete FileOutputStream class lets you write a stream of bytes to a file; the concrete FileInputStream class lets you read a stream of bytes from a file.
FileOutputStream subclasses OutputStream and declares five constructors for creating file output streams. For example, FileOutputStream(String name) creates a file output stream to the existing file identified by name. This constructor throws FileNotFoundException when the file doesn’t exist and cannot be created, it is a directory rather than a normal file, or there is some other reason why the file cannot be opened for output.

The following example uses FileOutputStream(String pathname) to create a file output stream with employee.dat as its destination:

FileOutputStream fos = new FileOutputStream("employee.dat");

**Tip** FileOutputStream(String name) overwrites an existing file. To append data instead of overwriting existing content, call a FileOutputStream constructor that includes a boolean append parameter and pass true to this parameter.

FileInputStream subclasses InputStream and declares three constructors for creating file input streams. For example, FileInputStream(String name) creates a file input stream from the existing file identified by name. This constructor throws FileNotFoundException when the file doesn’t exist, it is a directory rather than a normal file, or there is some other reason for why the file cannot be opened for input.

The following example uses FileInputStream(String name) to create a file input stream with employee.dat as its source:

FileInputStream fis = new FileInputStream("employee.dat");

FileOutputStream and FileInputStream are useful in a file-copying context. Listing 11-11 presents the source code to a Copy application that provides a demonstration.

**Listing 11-11. Copying a Source File to a Destination File**

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        FileInputStream fis = null;
        FileOutputStream fos = null;
```
try {
    fis = new FileInputStream(args[0]);
    fos = new FileOutputStream(args[1]);
    int b; // I chose b instead of byte because byte is a reserved word.
    while ((b = fis.read()) != -1)
        fos.write(b);
} catch (FileNotFoundException fnfe) {
    System.err.println(args[0] + " could not be opened for input, or " +
    args[1] + " could not be created for output");
} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    if (fis != null)
        try {
            fis.close();
        } catch (IOException ioe) {
            assert false; // shouldn't happen in this context
        }
    if (fos != null)
        try {
            fos.close();
        } catch (IOException ioe) {
            assert false; // shouldn't happen in this context
        }
}

Listing 11-11’s main() method first verifies that two command-line arguments, identifying the names of source and destination files, are specified. It then proceeds to instantiate FileInputStream and FileOutputStream and enter a while loop that repeatedly reads bytes from the file input stream and writes them to the file output stream.

Of course something might go wrong. Perhaps the source file doesn’t exist, or perhaps the destination file cannot be created (a same-named read-only file might exist, for example). In either scenario, FileNotFoundException is thrown and must be handled. Another possibility is that an I/O error occurred during the copy operation. Such an error results in IOException.
Regardless of an exception being thrown or not, the input and output streams are closed via the finally block. In a simple application like this, I could ignore the close() method calls and let the application terminate. Although Java automatically closes open files at this point, it’s good form to explicitly close files upon exit.

Because close() is capable of throwing an instance of the checked IOException class, a call to this method is wrapped in a try block with an appropriate catch block that catches this exception. Notice the if statement that precedes each try block. This statement is necessary to avoid a thrown NullPointerException instance should either fis or fos contain the null reference.

**PipedOutputStream and PipedInputStream**

Threads must often communicate. One approach involves using shared variables. Another approach involves using piped streams via the PipedOutputStream and PipedInputStream classes. The PipedOutputStream class lets a sending thread write a stream of bytes to an instance of the PipedInputStream class, which a receiving thread uses to subsequently read those bytes.

**Caution** Attempting to use a PipedOutputStream object and a PipedInputStream object from a single thread is not recommended because it might deadlock the thread.

PipedOutputStream declares a pair of constructors for creating piped output streams:

- PipedOutputStream() creates a piped output stream that’s not yet connected to a piped input stream. It must be connected to a piped input stream, either by the receiver or the sender, before being used.

- PipedOutputStream(PipedInputStream dest) creates a piped output stream that’s connected to piped input stream dest. Bytes written to the piped output stream can be read from dest. This constructor throws IOException when an I/O error occurs.

PipedOutputStream declares a void connect(PipedInputStream dest) method that connects this piped output stream to dest. This method throws IOException when this piped output stream is already connected to another piped input stream.

PipedInputStream declares four constructors for creating piped input streams:

- PipedInputStream() creates a piped input stream that’s not yet connected to a piped output stream. It must be connected to a piped output stream before being used.

- PipedInputStream(int pipeSize) creates a piped input stream that’s not yet connected to a piped output stream and uses pipeSize to size the piped input stream’s buffer. It must be connected to a piped output stream before being used. This constructor throws IllegalArgumentException when pipeSize is less than or equal to 0.
PipedInputStream(PipedOutputStream src) creates a piped input stream that's connected to piped output stream src. Bytes written to src can be read from this piped input stream. This constructor throws IOException when an I/O error occurs.

PipedInputStream(PipedOutputStream src, int pipeSize) creates a piped input stream that's connected to piped output stream src and uses pipeSize to size the piped input stream's buffer. Bytes written to src can be read from this piped input stream. This constructor throws IOException when an I/O error occurs and IllegalArgumentException when pipeSize is less than or equal to 0.

PipedInputStream declares a void connect(PipedOutputStream src) method that connects this piped input stream to src. This method throws IOException when this piped input stream is already connected to another piped output stream.

The easiest way to create a pair of piped streams is in the same thread and in either order. For example, you can first create the piped output stream.

```java
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream(pos);
```

Alternatively, you can first create the piped input stream.

```java
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

You can leave both streams unconnected and later connect them to each other using the appropriate piped stream's connect() method, as follows:

```java
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream();
// ...
pos.connect(pis);
```

Listing 11-12 presents a PipedStreamsDemo application whose sender thread streams a sequence of randomly generated byte integers to a receiver thread, which outputs this sequence.

**Listing 11-12. Piping Randomly Generated Bytes from a Sender Thread to a Receiver Thread**

```java
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class PipedStreamsDemo {
    public static void main(String[] args) throws IOException {
        final PipedOutputStream pos = new PipedOutputStream();
        final PipedInputStream pis = new PipedInputStream(pos);
```

```java
```
Runnable senderTask = new Runnable()
{
    final static int LIMIT = 10;

    @Override
    public void run()
    {
        try
        {
            for (int i = 0 ; i < LIMIT; i++)
                pos.write((byte) (Math.random() * 256));
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
    finally
    {
        try
        {
            pos.close();
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
};

Runnable receiverTask = new Runnable()
{
    @Override
    public void run()
    {
        try
        {
            int b;
            while ((b = pis.read()) != -1)
                System.out.println(b);
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
    finally
    {
        try
        {
            pis.close();
        }
    }
}
Listing 11-12’s main() method creates piped output and piped input streams that will be used by the senderTask thread to communicate a sequence of randomly generated byte integers and by the receiverTask thread to receive this sequence.

The sender task’s run() method explicitly closes its pipe stream when it finishes sending the data. If it didn’t do this, an IOException instance with a “write end dead” message would be thrown when the receiver thread invoked read() for the final time (which would otherwise return -1 to indicate end of stream). For more information on this message, check out Daniel Ferber’s “Whats this? IOException: Write end dead” blog post (http://techtavern.wordpress.com/2008/07/16/whats-this-ioexception-write-end-dead/).

Compile Listing 11-12 (javac PipedStreamsDemo.java) and run this application (java PipedStreamsDemo). You’ll discover output similar to the following:

93
23
125
50
126
131
210
29
150
91

**FilterOutputStream and FilterInputStream**

Byte array, file, and piped streams pass bytes unchanged to their destinations. Java also supports filter streams that buffer, compress/uncompress, encrypt/decrypt, or otherwise manipulate a stream’s byte sequence (that is input to the filter) before it reaches its destination.

A filter output stream takes the data passed to its write() methods (the input stream), filters it, and writes the filtered data to an underlying output stream, which might be another filter output stream or a destination output stream such as a file output stream.

Filter output streams are created from subclasses of the concrete FilterOutputStream class, an OutputStream subclass. FilterOutputStream declares a single FilterOutputStream(OutputStream out) constructor that creates a filter output stream built on top of out, the underlying output stream.
Listing 11-13 reveals that it’s easy to subclass FilterOutputStream. At minimum, you declare a constructor that passes its OutputStream argument to FilterOutputStream’s constructor and override FilterOutputStream’s write(int) method.

**Listing 11-13. Scrambling a Stream of Bytes**

```java
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;

public class ScrambledOutputStream extends FilterOutputStream {
    private int[] map;

    public ScrambledOutputStream(OutputStream out, int[] map) {
        super(out);
        if (map == null)
            throw new NullPointerException("map is null");
        if (map.length != 256)
            throw new IllegalArgumentException("map.length != 256");
        this.map = map;
    }

    @Override
    public void write(int b) throws IOException {
        out.write(map[b]);
    }
}
```

Listing 11-13 presents a ScrambledOutputStream class that performs trivial encryption on its input stream by scrambling the input stream’s bytes via a remapping operation. This constructor takes a pair of arguments:

- out identifies the output stream on which to write the scrambled bytes.
- map identifies an array of 256 byte-integer values to which input stream bytes map.

The constructor first passes its out argument to the FilterOutputStream parent via a super(out); call. It then verifies its map argument’s integrity (map must be nonnull and have a length of 256: a byte stream offers exactly 256 bytes to map) before saving map.

The write(int) method is trivial: it calls the underlying output stream’s write(int) method with the byte to which argument b maps. FilterOutputStream declares out to be protected (for performance), which is why I can directly access this field.

**Note** It’s only essential to override write(int) because FilterOutputStream’s other two write() methods are implemented via this method.
Listing 11-14 presents the source code to a Scramble application for experimenting with scrambling a source file's bytes via ScrambledOutputStream and writing these scrambled bytes to a destination file.

Listing 11-14. Scrambling a File's Bytes

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;

public class Scramble {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Scramble srcpath destpath");
            return;
        }
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream(args[0]);
            FileOutputStream fos = new FileOutputStream(args[1]);
            ScrambledOutputStream sos = new ScrambledOutputStream(fos, makeMap());
            int b;
            while ((b = fis.read()) != -1)
                sos.write(b);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
            }
            if (sos != null) {
                try {
                    sos.close();
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
            }
        }
    }
}
```
catch (IOException ioe)
{
    ioe.printStackTrace();
}
}
}

static int[] makeMap()
{
    int[] map = new int[256];
    for (int i = 0; i < map.length; i++)
        map[i] = i;
    // Shuffle map.
    Random r = new Random(0);
    for (int i = 0; i < map.length; i++)
    {
        int n = r.nextInt(map.length);
        int temp = map[i];
        map[i] = map[n];
        map[n] = temp;
    }
    return map;
}

Scramble’s main() method first verifies the number of command-line arguments: the first argument identifies the source path of the file with unscrambled content; the second argument identifies the destination path of the file that stores scrambled content.

Assuming that two command-line arguments have been specified, main() instantiates FileInputStream, creating a file input stream that’s connected to the file identified by args[0].

Continuing, main() instantiates FileOutputStream, creating a file output stream that’s connected to the file identified by args[1]. It then instantiates ScrambledOutputStream and passes the FileOutputStream instance to ScrambledOutputStream’s constructor.

    Note  When a stream instance is passed to another stream class’s constructor, the two streams are chained together. For example, the scrambled output stream is chained to the file output stream.

main() now enters a loop, reading bytes from the file input stream and writing them to the scrambled output stream by calling ScrambledOutputStream’s write(int) method. This loop continues until FileInputStream’s read() method returns -1 (end of file).

The finally block closes the file input stream and scrambled output stream by calling their close() methods. It doesn’t call the file output stream’s close() method because FilterOutputStream automatically calls the underlying output stream’s close() method.

The makeMap() method is responsible for creating the map array that’s passed to ScrambledOutputStream’s constructor. The idea is to populate the array with all 256 byte-integer values, storing them in random order.

www.it-ebooks.info
Suppose you have a simple 15-byte file named hello.txt that contains “Hello, World!” (followed by a carriage return and a line feed). If you execute `java Scramble hello.txt hello.out` on a Windows 7 platform, you’ll observe Figure 11-5’s scrambled output.

A filter input stream takes the data obtained from its underlying input stream—which might be another filter input stream or a source input stream such as a file input stream—filters it, and makes this data available via its `read()` methods (the output stream).

Filter input streams are created from subclasses of the concrete `FilterInputStream` class, an `InputStream` subclass. `FilterInputStream` declares a single `FilterInputStream(InputStream in)` constructor that creates a filter input stream built on top of `in`, the underlying input stream.

Listing 11-15 shows that it’s easy to subclass `FilterInputStream`. At minimum, declare a constructor that passes its `InputStream` argument to `FilterInputStream`'s constructor and override `FilterInputStream`'s `read()` and `read(byte[], int, int)` methods.

Listing 11-15. Unscrambling a Stream of Bytes

```java
import java.io.FilterInputStream;
import java.io.InputStream;
import java.io.IOException;

public class ScrambledInputStream extends FilterInputStream {
    private int[] map;
```
```java
public ScrambledInputStream(InputStream in, int[] map) {
    super(in);
    if (map == null)
        throw new NullPointerException("map is null");
    if (map.length != 256)
        throw new IllegalArgumentException("map.length != 256");
    this.map = map;
}

@Override
public int read() throws IOException {
    int value = in.read();
    return (value == -1) ? -1 : map[value];
}

@Override
public int read(byte[] b, int off, int len) throws IOException {
    int nBytes = in.read(b, off, len);
    if (nBytes <= 0)
        return nBytes;
    for (int i = 0; i < nBytes; i++)
        b[off + i] = (byte) map[off + i];
    return nBytes;
}
}
```

Listing 11-15 presents a ScrambledInputStream class that performs trivial decryption on its underlying input stream by unscrambling the underlying input stream's scrambled bytes via a remapping operation.

The read() method first reads the scrambled byte from its underlying input stream. If the returned value is -1 (end of file), this value is returned to its caller. Otherwise, the byte is mapped to its unscrambled value, which is returned.

The read(byte[], int, int) method is similar to read(), but stores bytes read from the underlying input stream in a byte array, taking an offset into this array and a length (number of bytes to read) into account.

Once again, -1 might be returned from the underlying read() method call. If so, this value must be returned. Otherwise, each byte in the array is mapped to its unscrambled value, and the number of bytes read is returned.

**Note** It's only essential to override read() and read(byte[], int, int) because FilterInputStream's read(byte[]) method is implemented via the latter method.
Listing 11-16 presents the source code to an Unscramble application for experimenting with ScrambledInputStream by unscrambling a source file’s bytes and writing these unscrambled bytes to a destination file.

Listing 11-16. Unscrambling a File’s Bytes

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;

public class Unscramble {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Unscramble srcpath destpath");
            return;
        }
        ScrambledInputStream sis = null;
        FileOutputStream fos = null;
        try {
            FileInputStream fis = new FileInputStream(args[0]);
            sis = new ScrambledInputStream(fis, makeMap());
            fos = new FileOutputStream(args[1]);
            int b;
            while ((b = sis.read()) != -1) {
                fos.write(b);
            }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        finally {
            if (sis != null) {
                try {
                    sis.close();
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
            }
            if (fos != null) {
                try {
                    fos.close();
                }
            }
        }
    }
}
```
```java
catch (IOException ioe)
{
    ioe.printStackTrace();
}

static int[] makeMap()
{
    int[] map = new int[256];
    for (int i = 0; i < map.length; i++)
        map[i] = i;
    // Shuffle map.
    Random r = new Random(0);
    for (int i = 0; i < map.length; i++)
    {
        int n = r.nextInt(map.length);
        int temp = map[i];
        map[i] = map[n];
        map[n] = temp;
    }
    int[] temp = new int[256];
    for (int i = 0; i < temp.length; i++)
        temp[map[i]] = i;
    return temp;
}
```

Unscramble's `main()` method first verifies the number of command-line arguments: the first argument identifies the source path of the file with scrambled content; the second argument identifies the destination path of the file that stores unscrambled content.

Assuming that two command-line arguments have been specified, `main()` instantiates `FileInputStream`, creating a file input stream that's connected to the file identified by `args[1]`. 

Continuing, `main()` instantiates `FileInputStream`, creating a file input stream that's connected to the file identified by `args[0]`. It then instantiates `ScrambledInputStream`, and passes the `FileInputStream` instance to `ScrambledInputStream`'s constructor.

Note When a stream instance is passed to another stream class’s constructor, the two streams are chained together. For example, the scrambled input stream is chained to the file input stream.

`main()` now enters a loop, reading bytes from the scrambled input stream and writing them to the file output stream. This loop continues until `ScrambledInputStream`'s `read()` method returns -1 (end of file).

The finally block closes the scrambled input stream and file output stream by calling their `close()` methods. It doesn't call the file input stream’s `close()` method because `FilterOutputStream` automatically calls the underlying input stream’s `close()` method.
The `makeMap()` method is responsible for creating the map array that's passed to `ScrambledInputStream`'s constructor. The idea is to duplicate Listing 11-14's map array and then invert it so that unscrambling can be performed.

Continuing from the previous `hello.txt/hello.out` example, execute `java Unscramble hello.out hello.bak` and you'll see the same unscrambled content in `hello.bak` that's present in `hello.txt`.

**Note** For an additional example of a filter output stream and its complementary filter input stream, check out the “Extending Java Streams to Support Bit Streams” article ([www.drdoobbs.com/184410423](http://www.drdoobbs.com/184410423)) on the Dr. Dobb's web site. This article introduces `BitStreamOutputStream` and `BitStreamInputStream` classes that are useful for outputting and inputting bit streams. The article then demonstrates these classes in a Java implementation of the Lempel-Zif-Welch (LZW) data compression and decompression algorithm.

### BufferedOutputStream and BufferedInputStream

FileOutputStream and FileInputStream have a performance problem. Each file output stream write() method call and file input stream read() method call results in a call to one of the underlying platform’s native methods, and these native calls slow down I/O.

**Note** A native method is an underlying platform API function that Java connects to an application via the Java Native Interface (JNI). Java supplies reserved word `native` to identify a native method. For example, the `RandomAccessFile` class declares a private native void `open(String name, int mode)` method. When a `RandomAccessFile` constructor calls this method, Java asks the underlying platform (via the JNI) to open the specified file in the specified mode on Java's behalf. I discuss the JNI in Chapter 16.

The concrete `BufferedOutputStream` and `BufferedInputStream` filter stream classes improve performance by minimizing underlying output stream write() and underlying input stream read() method calls. Instead, calls to `BufferedOutputStream`'s write() and `BufferedInputStream`'s read() methods take Java buffers into account.

- When a write buffer is full, write() calls the underlying output stream write() method to empty the buffer. Subsequent calls to `BufferedOutputStream`'s write() methods store bytes in this buffer until it's once again full.
- When the read buffer is empty, read() calls the underlying input stream read() method to fill the buffer. Subsequent calls to `BufferedInputStream`'s read() methods return bytes from this buffer until it's once again empty.

`BufferedOutputStream` declares the following constructors:

- `BufferedOutputStream(OutputStream out)` creates a buffered output stream that streams its output to `out`. An internal buffer is created to store bytes written to `out`.
- `BufferedOutputStream(OutputStream out, int size)` creates a buffered output stream that streams its output to `out`. An internal buffer of length `size` is created to store bytes written to `out`.

www.it-ebooks.info
The following example chains a `BufferedOutputStream` instance to a `FileOutputStream` instance. Subsequent `write()` method calls on the `BufferedOutputStream` instance buffer bytes and occasionally result in internal `write()` method calls on the encapsulated `FileOutputStream` instance.

```java
FileOutputStream fos = new FileOutputStream("employee.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos); // Chain bos to fos.
bos.write(0); // Write to employee.dat through the buffer.
// Additional write() method calls.
bos.close(); // This method call internally calls fos's close() method.
```

`BufferedInputStream` declares the following constructors:

- `BufferedInputStream(InputStream in)` creates a buffered input stream that streams its input from `in`. An internal buffer is created to store bytes read from `in`.
- `BufferedInputStream(InputStream in, int size)` creates a buffered input stream that streams its input from `in`. An internal buffer of length `size` is created to store bytes read from `in`.

The following example chains a `BufferedInputStream` instance to a `FileInputStream` instance. Subsequent `read()` method calls on the `BufferedInputStream` instance unbuffer bytes and occasionally result in internal `read()` method calls on the encapsulated `FileInputStream` instance.

```java
FileInputStream fis = new FileInputStream("employee.dat");
BufferedInputStream bis = new BufferedInputStream(fis); // Chain bis to fis.
ing ch = bis.read(); // Read employee.dat through the buffer.
// Additional read() method calls.
bis.close(); // This method call internally calls fis's close() method.
```

**DataOutputStream and DataInputStream**

`FileOutputStream` and `FileInputStream` are useful for writing and reading bytes and arrays of bytes. However, they provide no support for writing and reading primitive type values (such as integers) and strings.

For this reason, Java provides the concrete `DataOutputStream` and `DataInputStream` filter stream classes. Each class overcomes this limitation by providing methods to write or read primitive type values and strings in a platform-independent way.

- Integer values are written and read in *big-endian format* (the most significant byte comes first). Check out Wikipedia’s “Endianness” entry [http://en.wikipedia.org/wiki/Endianness](http://en.wikipedia.org/wiki/Endianness) to learn about the concept of endianness.
- Floating-point and double precision floating-point values are written and read according to the IEEE 754 standard, which specifies 4 bytes per floating-point value and 8 bytes per double precision floating-point value.
- Strings are written and read according to a modified version of *UTF-8*, a variable-length encoding standard for efficiently storing 2-byte Unicode characters. Check out Wikipedia’s “UTF-8” entry [http://en.wikipedia.org/wiki/Utf-8](http://en.wikipedia.org/wiki/Utf-8) to learn more about UTF-8.
DataOutputStream declares a single DataOutputStream(OutputStream out) constructor. Because this class implements the DataOutput interface, DataOutputStream also provides access to the same-named write methods as provided by RandomAccessFile.

DataInputStream declares a single DataInputStream(InputStream in) constructor. Because this class implements the DataInput interface, DataInputStream also provides access to the same-named read methods as provided by RandomAccessFile.

Listing 11-17 presents the source code to a DataStreamsDemo application that uses a DataOutputStream instance to write multibyte values to a FileOutputStream instance and uses a DataInputStream instance to read multibyte values from a FileInputStream instance.

Listing 11-17. Outputting and Then Inputting a Stream of Multibyte Values

```java
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataStreamsDemo
{
    final static String FILENAME = "values.dat";

    public static void main(String[] args)
    {
        DataOutputStream dos = null;
        DataInputStream dis = null;
        try
        {
            FileOutputStream fos = new FileOutputStream(FILENAME);
            dos = new DataOutputStream(fos);
            dos.writeInt(1995);
            dos.writeUTF("Saving this String in modified UTF-8 format!");
            dos.writeFloat(1.0F);
            dos.close(); // Close underlying file output stream.
            // The following null assignment prevents another close attempt on
            // dos (which is now closed) should IOException be thrown from
            // subsequent method calls.
            dos = null;
            FileInputStream fis = new FileInputStream(FILENAME);
            dis = new DataInputStream(fis);
            System.out.println(dis.readInt());
            System.out.println(dis.readUTF());
            System.out.println(dis.readFloat());
        }
        catch (IOException ioe)
        {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }

```
finally
{
    if (dos != null)
    try
    {
        dos.close();
    }
    catch (IOException ioe2) // Cannot redeclare local variable ioe.
    {
        assert false; // shouldn't happen in this context
    }
    if (dis != null)
    try
    {
        dis.close();
    }
    catch (IOException ioe2) // Cannot redeclare local variable ioe.
    {
        assert false; // shouldn't happen in this context
    }
}

DataStreamsDemo creates a file named values.dat; calls DataOutputStream methods to write an integer, a string, and a floating-point value to this file; and calls DataInputStream methods to read back these values. Unsurprisingly, it generates the following output:

1995
Saving this String in modified UTF-8 format!
1.0

**Caution** When reading a file of values written by a sequence of DataOutputStream method calls, make sure to use the same method-call sequence. Otherwise, you’re bound to end up with erroneous data and, in the case of the readUTF() methods, thrown instances of the java.io.UTFDataFormatException class (a subclass of IOException).

**Object Serialization and Deserialization**

Java provides the DataOutputStream and DataInputStream classes to stream primitive type values and String objects. However, you cannot use these classes to stream non-String objects. Instead, you must use object serialization and deserialization to stream objects of arbitrary types.

*Object serialization* is a virtual machine mechanism for *serializing* object state into a stream of bytes. Its *deserialization* counterpart is a virtual machine mechanism for *deserializing* this state from a byte stream.
Note  An object’s state consists of instance fields that store primitive-type values and/or references to other objects. When an object is serialized, the objects that are part of this state are also serialized (unless you prevent them from being serialized). Furthermore, the objects that are part of those objects’ states are serialized (unless you prevent this), and so on.

Java supports default serialization and deserialization, custom serialization and deserialization, and externalization.

Default Serialization and Deserialization

Default serialization and deserialization is the easiest form to use but offers little control over how objects are serialized and deserialized. Although Java handles most of the work on your behalf, there are a couple of tasks that you must perform.

Your first task is to have the class of the object that’s to be serialized implement the `java.io.Serializable` interface, either directly or indirectly via the class’s superclass. The rationale for implementing `Serializable` is to avoid unlimited serialization.

Note  `Serializable` is an empty marker interface (there are no methods to implement) that a class implements to tell the virtual machine that it’s okay to serialize the class’s objects. When the serialization mechanism encounters an object whose class doesn’t implement `Serializable`, it throws an instance of the `java.io.NotSerializableException` class (an indirect subclass of `IOException`).

Unlimited serialization is the process of serializing an entire object graph. Java doesn’t support unlimited serialization for the following reasons:

- **Security**: If Java automatically serialized an object containing sensitive information (such as a password or a credit card number), it would be easy for a hacker to discover this information and wreak havoc. It’s better to give the developer a choice to prevent this from happening.

- **Performance**: Serialization leverages the Reflection API (discussed in Chapter 8), which tends to slow down application performance. Unlimited serialization could really hurt an application’s performance.

- **Objects not amenable to serialization**: Some objects exist only in the context of a running application and it’s meaningless to serialize them. For example, a file stream object that’s deserialized no longer represents a connection to a file.

Listing 11-18 declares an `Employee` class that implements the `Serializable` interface to tell the virtual machine that it’s okay to serialize `Employee` objects.
Listing 11-18. Implementing Serializable

```java
import java.io.Serializable;

public class Employee implements Serializable
{
    private String name;
    private int age;

    public Employee(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }

    public int getAge() { return age; }
}
```

Because `Employee` implements `Serializable`, the serialization mechanism will not throw a `NotSerializableException` instance when serializing an `Employee` object. Not only does `Employee` implement `Serializable`, the `String` class also implements this interface.

Your second task is to work with the `ObjectOutputStream` class and its `writeObject()` method to serialize an object and the `ObjectInputStream` class and its `readObject()` method to deserialize the object.

Note: Although `ObjectOutputStream` extends `OutputStream` instead of `FilterOutputStream`, and although `ObjectInputStream` extends `InputStream` instead of `FilterInputStream`, these classes behave as filter streams.

Java provides the concrete `ObjectOutputStream` class to initiate the serialization of an object’s state to an object output stream. This class declares an `ObjectOutputStream(OutputStream out)` constructor that chains the object output stream to the output stream specified by `out`.

When you pass an output stream reference to `out`, this constructor attempts to write a serialization header to that output stream. It throws `NullPointerException` when `out` is null and `IOException` when an I/O error prevents it from writing this header.

`ObjectOutputStream` serializes an object via its `void writeObject(Object obj)` method. This method attempts to write information about `obj`’s class followed by the values of `obj`’s instance fields to the underlying output stream.

`writeObject()` doesn’t serialize the contents of static fields. In contrast, it serializes the contents of all instance fields that are not explicitly prefixed with the `transient` reserved word. For example, consider the following field declaration:

```
public transient char[] password;
```
This declaration specifies transient to avoid serializing a password for some hacker to encounter. The virtual machine's serialization mechanism ignores any instance field that's marked transient.

**Note** Check out my “Transience” blog post ([www.javaworld.com/community/node/13451](http://www.javaworld.com/community/node/13451)) to learn more about transient.

writeObject() throws IOException or an instance of an IOException subclass when something goes wrong. For example, this method throws NotSerializableException when it encounters an object whose class doesn't implement Serializable.

**Note** Because ObjectOutputStream implements DataOutput, it also declares methods for writing primitive-type values and strings to an object output stream.

Java provides the concrete ObjectInputStream class to initiate the deserialization of an object's state from an object input stream. This class declares an ObjectInputStream(InputStream in) constructor that chains the object input stream to the input stream specified by in.

When you pass an input stream reference to in, this constructor attempts to read a serialization header from that input stream. It throws NullPointerException when in is null, IOException when an I/O error prevents it from reading this header, and java.io.StreamCorruptedException (an indirect subclass of IOException) when the stream header is incorrect.

ObjectInputStream deserializes an object via its Object readObject() method. This method attempts to read information about obj's class followed by the values of obj's instance fields from the underlying input stream.

readObject() throws java.lang.ClassNotFoundException, IOException, or an instance of an IOException subclass when something goes wrong. For example, this method throws java.io.OptionalDataException when it encounters primitive-type values instead of objects.

**Note** Because ObjectInputStream implements DataInput, it also declares methods for reading primitive-type values and strings from an object input stream.

Listing 11-19 presents an application that uses these classes to serialize and deserialize an instance of Listing 11-18's Employee class to and from an employee.dat file.

**Listing 11-19. Serializing and Deserializing an Employee Object**

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

www.it-ebooks.info
public class SerializationDemo
{
    final static String FILENAME = "employee.dat";

    public static void main(String[] args)
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try
        {
            FileOutputStream fos = new FileOutputStream(FILENAME);
            oos = new ObjectOutputStream(fos);
            Employee emp = new Employee("John Doe", 36);
            oos.writeObject(emp);
            oos.close();
            oos = null;
            FileInputStream fis = new FileInputStream(FILENAME);
            ois = new ObjectInputStream(fis);
            emp = (Employee) ois.readObject(); // (Employee) cast is necessary.
            ois.close();
            System.out.println(emp.getName());
            System.out.println(emp.getAge());
        }
        catch (ClassNotFoundException cnfe)
        {
            System.err.println(cnfe.getMessage());
        }
        catch (IOException ioe)
        {
            System.err.println(ioe.getMessage());
        }
        finally
        {
            if (oos != null)
            try
            {
                oos.close();
            }
            catch (IOException ioe)
            {
                assert false; // shouldn't happen in this context
            }
            if (ois != null)
            try
            {
                ois.close();
            }
        }
    }
}
catch (IOException ioe)
{
    assert false; // shouldn't happen in this context
}
}

Listing 11-19’s main() method first instantiates Employee and serializes this instance via writeObject() to employee.dat. It then deserializes this instance from this file via readObject() and invokes the instance’s getName() and getAge() methods. Along with employee.dat, you’ll discover the following output when you run this application:

```
John Doe
36
```

There’s no guarantee that the same class will exist when a serialized object is deserialized (perhaps an instance field has been deleted). During deserialization, this mechanism causes readObject() to throw java.io.InvalidClassException—an indirect subclass of the IOException class—when it detects a difference between the deserialized object and its class.

Every serialized object has an identifier. The deserialization mechanism compares the identifier of the object being deserialized with the serialized identifier of its class (all serializable classes are automatically given unique identifiers unless they explicitly specify their own identifiers) and causes InvalidClassException to be thrown when it detects a mismatch.

Perhaps you’ve added an instance field to a class, and you want the deserialization mechanism to set the instance field to a default value rather than have readObject() throw an InvalidClassException instance. (The next time you serialize the object, the new field’s value will be written out.)

You can avoid the thrown InvalidClassException instance by adding a static final long serialVersionUID = long integer value; declaration to the class. The long integer value must be unique and is known as a stream unique identifier (SUID).

During deserialization, the virtual machine will compare the deserialized object’s SUID to its class’s SUID. If they match, readObject() will not throw InvalidClassException when it encounters a compatible class change (such as adding an instance field). However, it will still throw this exception when it encounters an incompatible class change (such as changing an instance field’s name or type).

```
Note Whenever you change a class in some fashion, you must calculate a new SUID and assign it to serialVersionUID.
```
The JDK provides a `serialver` tool for calculating the SUID. For example, to generate an SUID for Listing 11-18's `Employee` class, change to the directory containing `Employee.class` and execute the following command:

```bash
serialver Employee
```

In response, `serialver` generates the following output, which you paste (except for `Employee:`) into `Employee.java`:

```java
Employee:    static final long serialVersionUID = 1517331364702470316L;
```

The Windows version of `serialver` also provides a graphical user interface that you might find more convenient to use. To access this interface, specify the following command line:

```bash
serialver -show
```

When the `serialver` window appears, enter `Employee` into the Full Class Name text field and click the Show button, as demonstrated in Figure 11-6.

![Figure 11-6. The serialver user interface reveals Employee's SUID](image)

**Custom Serialization and Deserialization**

My previous discussion focused on default serialization and deserialization (with the exception of marking an instance field `transient` to prevent it from being included during serialization). However, situations arise where you need to customize these tasks.

For example, suppose you want to serialize instances of a class that doesn’t implement `Serializable`. As a workaround, you subclass this other class, have the subclass implement `Serializable`, and forward subclass constructor calls to the superclass.

Although this workaround lets you serialize subclass objects, you cannot deserialize these serialized objects when the superclass doesn’t declare a noargument constructor, which is required by the deserialization mechanism. Listing 11-20 demonstrates this problem.

**Listing 11-20. Problematic Deserialization**

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
```

www.it-ebooks.info
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Employee
{
    private String name;

    Employee(String name)
    {
        this.name = name;
    }

    @Override
    public String toString()
    {
        return name;
    }
}

class SerEmployee extends Employee implements Serializable
{
    SerEmployee(String name)
    {
        super(name);
    }
}

public class SerializationDemo
{
    public static void main(String[] args)
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try
        {
            oos = new ObjectOutputStream(new FileOutputStream("employee.dat"));
            SerEmployee se = new SerEmployee("John Doe");
            System.out.println(se);
            oos.writeObject(se);
            oos.close();
            oos = null;
            System.out.println("se object written to file");
            ois = new ObjectInputStream(new FileInputStream("employee.dat"));
            se = (SerEmployee) ois.readObject();
            System.out.println("se object read from file");
            System.out.println(se);
        }
        catch (ClassNotFoundException cnfe)
        {
            cnfe.printStackTrace();
        }
    }
}
catch (IOException ioe)  
{
   ioe.printStackTrace();
}
finally
{
   if (oos != null)
      try
      {
         oos.close();
      }
      catch (IOException ioe)
      {
         assert false; // shouldn't happen in this context
      }
   if (ois != null)
      try
      {
         ois.close();
      }
      catch (IOException ioe)
      {
         assert false; // shouldn't happen in this context
      }
} 

Listing 11-20’s main() method instantiates SerEmployee with an employee name. This class’s SerEmployee(String) constructor passes this argument to its Employee counterpart.

main() next calls Employee’s toString() method indirectly via System.out.println() to obtain this name, which is then output.

Continuing, main() serializes the SerEmployee instance to an employee.dat file via writeObject(). It then attempts to deserialize this object via readObject(), and this is where the trouble occurs, as revealed by the following output:

John Doe
se object written to file
java.io.InvalidClassException: SerEmployee; no valid constructor
   at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(Unknown Source)
   at java.io.ObjectStreamClass.checkDeserialize(Unknown Source)
   at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
   at java.io.ObjectInputStream.readObject0(Unknown Source)
   at java.io.ObjectInputStream.readObject(Unknown Source)
   at SerializationDemo.main(SerializationDemo.java:48)

This output reveals a thrown instance of the InvalidClassException class. This exception object was thrown during deserialization because Employee doesn't possess a noargument constructor.
You can overcome this problem by taking advantage of the wrapper class pattern that I presented in Chapter 4. Furthermore, you declare a pair of private methods in the subclass that the serialization and deserialization mechanisms look for and call.

Normally, the serialization mechanism writes out a class’s instance fields to the underlying output stream. However, you can prevent this from happening by declaring a private void 
\texttt{writeObject(ObjectOutputStream oos)} method in that class.

When the serialization mechanism discovers this method, it calls the method instead of automatically outputting instance field values. The only values that are output are those explicitly output via the method.

Conversely, the deserialization mechanism assigns values to a class’s instance fields that it reads from the underlying input stream. However, you can prevent this from happening by declaring a private void \texttt{readObject(ObjectInputStream ois)} method.

When the deserialization mechanism discovers this method, it calls the method instead of automatically assigning values to instance fields. The only values that are assigned to instance fields are those explicitly assigned via the method.

Because \texttt{SerEmployee} doesn’t introduce any fields, and because \texttt{Employee} doesn’t offer access to its internal fields (assume you don’t have the source code for this class), what would a serialized \texttt{SerEmployee} object include?

Although you cannot serialize \texttt{Employee}’s internal state, you can serialize the argument(s) passed to its constructors, such as the employee name.

Listing 11-21 reveals the refactored \texttt{SerEmployee} and \texttt{SerializationDemo} classes.

\textbf{Listing 11-21. Solving Problematic Deserialization}

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Employee {
    private String name;

    Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```
class SerEmployee implements Serializable {
    private Employee emp;
    private String name;

    SerEmployee(String name) {
        this.name = name;
        emp = new Employee(name);
    }

    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.writeUTF(name);
    }

    private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException {
        name = ois.readUTF();
        emp = new Employee(name);
    }

    @Override
    public String toString() {
        return name;
    }
}

public class SerializationDemo {
    public static void main(String[] args) {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("employee.dat");
            SerEmployee se = new SerEmployee("John Doe");
            System.out.println(se);
            oos.writeObject(se);
            oos.close();
            oos = null;
            System.out.println("se object written to file");
            ois = new ObjectInputStream(new FileInputStream("employee.dat");
            se = (SerEmployee) ois.readObject();
            System.out.println("se object read from file");
            System.out.println(se);
        } catch (ClassNotFoundException cnfe) {
            cnfe.printStackTrace();
        }
    }
}
catch (IOException ioe)
{
    ioe.printStackTrace();
}

finally
{
    if (oos != null)
        try
        {
            oos.close();
        }
        catch (IOException ioe)
        {
            assert false; // shouldn't happen in this context
        }
    if (ois != null)
        try
        {
            ois.close();
        }
        catch (IOException ioe)
        {
            assert false; // shouldn't happen in this context
        }
}

```

SerEmployee's writeObject() and readObject() methods rely on DataOutput and DataInput methods: they don't need to call ObjectOutputStream's writeObject() method and ObjectInputStream's readObject() method to perform their tasks.

When you run this application, it generates the following output:

```

John Doe
se object written to file
se object read from file
John Doe

The writeObject() and readObject() methods can be used to serialize/deserialize data items beyond the normal state (non-transient instance fields), for example, serializing/deserializing the contents of a static field.

However, before serializing or deserializing the additional data items, you must tell the serialization and deserialization mechanisms to serialize or deserialize the object's normal state. The following methods help you accomplish this task:

- ObjectOutputStream's defaultWriteObject() method outputs the object's normal state. Your writeObject() method first calls this method to output that state and then outputs additional data items via ObjectOutputStream methods such as writeUTF().
ObjectInputStream's `defaultReadObject()` method inputs the object's normal state. Your `readObject()` method first calls this method to input that state and then inputs additional data items via `ObjectInputStream` methods such as `readUTF()`.

**Externalization**

Along with default serialization/deserialization and custom serialization/deserialization, Java supports externalization. Unlike default/custom serialization/deserialization, *externalization* offers complete control over the serialization and deserialization tasks.

### Note

Externalization helps you improve the performance of the reflection-based serialization and deserialization mechanisms by giving you complete control over what fields are serialized and deserialized.

Java supports externalization via `java.io.Externalizable`. This interface declares the following pair of public methods:

- `void writeExternal(ObjectOutput out)` saves the calling object's contents by calling various methods on the `out` object. This method throws `IOException` when an I/O error occurs. (`java.io.ObjectOutput` is a subinterface of `DataOutput` and is implemented by `ObjectOutputStream`.)

- `void readExternal(ObjectInput in)` restores the calling object's contents by calling various methods on the `in` object. This method throws `IOException` when an I/O error occurs and `ClassNotFoundException` when the class of the object being restored cannot be found. (`java.io.ObjectInput` is a subinterface of `DataInput` and is implemented by `ObjectInputStream`.)

If a class implements `Externalizable`, its `writeExternal()` method is responsible for saving all field values that are to be saved. Also, its `readExternal()` method is responsible for restoring all saved field values and in the order they were saved.

Listing 11-22 presents a refactored version of Listing 11-18's `Employee` class to show you how to take advantage of externalization.

**Listing 11-22. Refactoring Listing 11-18's `Employee` Class to Support Externalization**

```java
class Employee implements Externalizable {
    private String name;
    private int age;
    
    // Implement writeExternal() and readExternal() methods here.
}
```

```java
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Employee implements Externalizable {
    private String name;
    private int age;
    
    // Implement writeExternal() and readExternal() methods here.
}
```
public Employee()
{
    System.out.println("Employee() called");
}

public Employee(String name, int age)
{
    this.name = name;
    this.age = age;
}

public String getName() { return name; }

public int getAge() { return age; }

@Override
public void writeExternal(ObjectOutput out) throws IOException
{
    System.out.println("writeExternal() called");
    out.writeUTF(name);
    out.writeInt(age);
}

@Override
public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
{
    System.out.println("readExternal() called");
    name = in.readUTF();
    age = in.readInt();
}

Employee declares a public Employee() constructor because each class that participates in externalization must declare a public noargument constructor. The deserialization mechanism calls this constructor to instantiate the object.

Caution The deserialization mechanism throws InvalidClassException with a “no valid constructor” message when it doesn’t detect a public noargument constructor.

Initiate externalization by instantiating ObjectOutputStream and calling its writeObject(Object) method, or by instantiating ObjectInputStream and calling its readObject() method.

Note When passing an object whose class (directly/indirectly) implements Externalizable to writeObject(), the writeObject()-initiated serialization mechanism writes only the identity of the object’s class to the object output stream.
Suppose you compiled Listing 11-19's SerializationDemo.java source code and Listing 11-22's Employee.java source code in the same directory. Now suppose you executed java SerializationDemo. In response, you would observe the following output:

writeExternal() called
Employee() called
readExternal() called
John Doe
36

Before serializing an object, the serialization mechanism checks the object's class to see if it implements Externalizable. If so, the mechanism calls writeExternal(). Otherwise, it looks for a private writeObject(ObjectOutputStream) method and calls this method when present. When this method isn't present, this mechanism performs default serialization, which includes only non-transient instance fields.

Before deserializing an object, the deserialization mechanism checks the object's class to see if it implements Externalizable. If so, the mechanism attempts to instantiate the class via the public no-argument constructor. Assuming success, it calls readExternal().

When the object's class doesn't implement Externalizable, the deserialization mechanism looks for a private readObject(ObjectInputStream) method. When this method isn't present, this mechanism performs default deserialization, which includes only non-transient instance fields.

**PrintStream**

Of all the stream classes, PrintStream is an oddball: it should have been named PrintOutputStream for consistency with the naming convention. This filter output stream class writes string representations of input data items to the underlying output stream.

---

**Note**  
PrintStream uses the default character encoding to convert a string's characters to bytes. (I'll discuss character encodings when I introduce you to writers and readers in the next section.) Because PrintStream doesn't support different character encodings, you should use the equivalent PrintWriter class instead of PrintStream. However, you need to know about PrintStream because of Standard I/O (see Chapter 1 for an introduction to this topic).

---

PrintStream instances are print streams whose various print() and println() methods print string representations of integers, floating-point values, and other data items to the underlying output stream. Unlike the print() methods, println() methods append a line terminator to their output.
The line terminator (also known as line separator) isn’t necessarily the newline (also commonly referred to as line feed). Instead, to promote portability, the line separator is the sequence of characters defined by system property `line.separator`. On Windows platforms, `System.getProperty("line.separator")` returns the actual carriage return code (13), which is symbolically represented by \r, followed by the actual newline/line feed code (10), which is symbolically represented by \n. In contrast, `System.getProperty("line.separator")` returns only the actual newline/line feed code on Unix and Linux platforms.

The `println()` methods call their corresponding `print()` methods followed by the equivalent of the `void println()` method, which eventually results in `line.separator`'s value being output. For example, `void println(int x)` outputs `x`’s string representation and calls this method to output the line separator.

Never hard-code the \n escape sequence in a string literal that you are going to output via a `print()` or `println()` method. Doing so isn’t portable. For example, when Java executes `System.out.print("first line\n");` followed by `System.out.println("second line");`, you will see `first line` on one line followed by `second line` on a subsequent line when this output is viewed at the Windows command line. In contrast, you’ll see `first line` followed by `second line` when this output is viewed in the Windows Notepad application (which requires a carriage return/line feed sequence to terminate lines). When you need to output a blank line, the easiest way to do this is to call `System.out.println();`, which is why you find this method call used elsewhere in my book. I confess that I don’t always follow my own advice, so you might find instances of \n in literal strings being passed to `System.out.print()` or `System.out.println()` elsewhere in this book.

PrintStream offers three other features that you’ll find useful:

- Unlike other output streams, a print stream never rethrows an `IOException` instance thrown from the underlying output stream. Instead, exceptional situations set an internal flag that can be tested by calling `PrintStream`’s boolean `checkError()` method, which returns true to indicate a problem.

- `PrintStream` objects can be created to automatically flush their output to the underlying output stream. In other words, the `flush()` method is automatically called after a byte array is written, one of the `println()` methods is called, or a newline is written.

- `PrintStream` declares a `PrintStream format(String format, Object... args)` method for achieving formatted output. Behind the scene, this method works with the `Formatter` class that I introduce in Chapter 13. `PrintStream` also declares a `printf(String format, Object... args)` convenience method that delegates to the `format()` method. For example, invoking `printf()` via `out.printf(format, args)` is identical to invoking `out.format(format, args).`
Standard I/O Revisited

In Chapter 1, I introduced you to Standard I/O. I stated that you input data items from the standard input stream by making System.in.read() method calls, that you output data items to the standard output stream by making System.out.print() and System.out.println() method calls, and that you output data items to the standard error stream by making System.err.print() and System.err.println() method calls. Finally, I discussed I/O redirection.

System.in, System.out, and System.err are formally described by the following class fields in the System class:

- public static final InputStream in
- public static final PrintStream out
- public static final PrintStream err

These fields contain references to InputStream and PrintStream objects that represent the standard input, standard output, and standard error streams.

When you invoke System.in.read(), the input is originating from the source identified by the InputStream instance assigned to in. Similarly, when you invoke System.out.print() or System.err.println(), the output is being sent to the destination identified by the PrintStream instance assigned to out or err, respectively.

**Tip** On an Android device, you can view content sent to standard output and standard error by first executing adb logcat at the command line. adb is one of the tools included in the Android SDK.

Java initializes in to refer to the keyboard or a file when the standard input stream is redirected to the file. Similarly, Java initializes out/err to refer to the screen or a file when the standard output/error stream is redirected to the file. You can programmatically specify the input source, output destination, and error destination by calling the following System class methods:

- void setIn(InputStream in)
- void setOut(PrintStream out)
- void setErr(PrintStream err)

Listing 11-23 presents an application that shows you how to use these methods to programmatically redirect the standard input, standard output, and standard error destinations.

**Listing 11-23. Programatically Specifying the Standard Input Source and Standard Output/Error Destinations**

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

public class RedirectIO {
    public static void main(String[] args) throws IOException {
        // Set standard input to a file
        FileInputStream in = new FileInputStream("input.txt");
        System.setIn(in);

        // Set standard output to a file
        PrintStream out = new PrintStream("output.txt");
        System.setOut(out);

        // Set standard error to a file
        PrintStream err = new PrintStream("error.txt");
        System.setErr(err);

        // ... program execution ...
    }
}
```
CHAPTER 11: Performing Classic I/O

if (args.length != 3)
{
    System.err.println("usage: java RedirectIO stdinfile stdoutfile stderrfile");
    return;
}

System.setIn(new FileInputStream(args[0]));
System.setOut(new PrintStream(args[1]));
System.setErr(new PrintStream(args[2]));

int ch;
while ((ch = System.in.read()) != -1)
    System.out.print((char) ch);

System.err.println("Redirected error output");
}

Listing 11-23 presents a RedirectIO application that lets you specify (via command-line arguments) the name of a file from which System.in.read() obtains its content as well as the names of files to which System.out.print() and System.err.println() send their content. It then proceeds to copy standard input to standard output and then demonstrates outputting content to standard error.

Next, new FileInputStream(args[0]) provides access to the input sequence of bytes that is stored in the file identified by args[0]. Similarly, new PrintStream(args[1]) provides access to the file identified by args[1], which will store the output sequence of bytes, and new PrintStream(args[2]) provides access to the file identified by args[2], which will store the error sequence of bytes.

Compile Listing 11-23 (javac RedirectIO.java). Then execute the following command line:

java RedirectIO RedirectIO.java out.txt err.txt

This command line produces no visual output on the screen. Instead, it copies the contents of RedirectIO.java to out.txt. It also stores Redirected error output in err.txt.

Working with Writers and Readers

Java’s stream classes are good for streaming sequences of bytes, but they’re not good for streaming sequences of characters because bytes and characters are two different things: a byte represents an 8-bit data item and a character represents a 16-bit data item. Also, Java’s char and String types naturally handle characters instead of bytes.

More importantly, byte streams have no knowledge of character sets (sets of mappings between integer values, known as code points, and symbols, such as Unicode) and their character encodings (mappings between the members of a character set and sequences of bytes that encode these characters for efficiency, such as UTF-8).

If you need to stream characters, you should take advantage of Java’s writer and reader classes, which were designed to support character I/O (they work with char instead of byte). Furthermore, the writer and reader classes take character encodings into account.
Early computers and programming languages were created mainly by English-speaking programmers in countries where English was the native language. They developed a standard mapping between code points 0 through 127, and the 128 commonly used characters in the English language (such as A–Z). The resulting character set/encoding was named American Standard Code for Information Interchange (ASCII).

The problem with ASCII is that it’s inadequate for most non-English languages. For example, ASCII doesn’t support diacritical marks such as the cedilla used in French. Because a byte can represent a maximum of 256 different characters, developers around the world started creating different character sets/encodings that encoded the 128 ASCII characters, but also encoded extra characters to meet the needs of languages such as French, Greek, or Russian. Over the years, many legacy (and still important) data files have been created whose bytes represent characters defined by specific character sets/encodings.

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have worked to standardize these 8-bit character sets/encodings under a joint umbrella standard called ISO/IEC 8859. The result is a series of substandards named ISO/IEC 8859-1, ISO/IEC 8859-2, and so on. For example, ISO/IEC 8859-1 (also known as Latin-1) defines a character set/encoding that consists of ASCII plus the characters covering most Western European countries. Also, ISO/IEC 8859-2 (also known as Latin-2) defines a similar character set/encoding covering Central and Eastern European countries.

Despite ISO’s/IEC’s best efforts, a plethora of character sets/encodings is still inadequate. For example, most character sets/encodings only allow you to create documents in a combination of English and one other language (or a small number of other languages). You cannot, for example, use an ISO/IEC character set/encoding to create a document using a combination of English, French, Turkish, Russian, and Greek characters.

This and other problems are being addressed by an international effort that has created and is continuing to develop Unicode, a single universal character set. Because Unicode characters are bigger than ISO/IEC characters, Unicode uses one of several variable-length encoding schemes known as Unicode Transformation Format (UTF) to encode Unicode characters for efficiency. For example, UTF-8 encodes every character in the Unicode character set in 1 to 4 bytes (and is backward compatible with ASCII).

The terms character set and character encoding are often used interchangeably. They mean the same thing in the context of ISO/IEC character sets in which a code point is the encoding. However, these terms are different in the context of Unicode in which Unicode is the character set and UTF-8 is one of several possible character encodings for Unicode characters.

**Writer and Reader Classes Overview**

The java.io package provides several writer and reader classes that are descendants of the abstract Writer and Reader classes. Figure 11-7 reveals the hierarchy of writer classes.
Figure 11-7. Unlike FilterOutputStream, FilterWriter is abstract

Figure 11-8 reveals the hierarchy of reader classes.

Figure 11-8. Unlike FilterInputStream, FilterReader is abstract

Although the writer and reader class hierarchies are similar to their output stream and input stream counterparts, there are differences. For example, FilterWriter and FilterReader are abstract, whereas their FilterOutputStream and FilterInputStream equivalents are not abstract. Also, BufferedWriter and BufferedReader don't extend FilterWriter and FilterReader, whereas BufferedOutputStream and BufferedInputStream extend FilterOutputStream and FilterInputStream.
The output stream and input stream classes were introduced in Java 1.0. After their release, design issues emerged. For example, `FilterOutputStream` and `FilterInputStream` should have been abstract. However, it was too late to make these changes because the classes were already being used; making these changes would have resulted in broken code. The designers of Java 1.1’s writer and reader classes took the time to correct these mistakes.

**Note** Regarding `BufferedWriter` and `BufferedReader` directly subclassing `Writer` and `Reader` instead of `FilterWriter` and `FilterReader`, I believe that this change has to do with performance. Calls to `BufferedOutputStream`'s `write()` methods and `BufferedInputStream`'s `read()` methods result in calls to `FilterOutputStream`'s `write()` methods and `FilterInputStream`'s `read()` methods. Because a file I/O activity such as copying one file to another can involve many `write()`/`read()` method calls, you want the best performance possible. By not subclassing `FilterWriter` and `FilterReader`, `BufferedWriter` and `BufferedReader` achieve better performance.

For brevity, I focus only on the `Writer`, `Reader`, `OutputStreamWriter`, `OutputStreamReader`, `FileWriter`, and `FileReader` classes in this chapter.

**Writer and Reader**

Java provides the `Writer` and `Reader` classes for performing character I/O. `Writer` is the superclass of all writer subclasses. The following list identifies differences between `Writer` and `OutputStream`:

- Writer declares several `append()` methods for appending characters to this writer. These methods exist because `Writer` implements the `java.lang.Appendable` interface, which is used in partnership with the `Formatter` class (discussed in Chapter 13) to output formatted strings.

- Writer declares additional `write()` methods, including a convenient `void write(String str)` method for writing a `String` object’s characters to this writer.

`Reader` is the superclass of all reader subclasses. The following list identifies differences between `Reader` and `InputStream`:

- Reader declares `read(char[])` and `read(char[], int, int)` methods instead of `read(byte[])` and `read(byte[], int, int)` methods.

- Reader doesn’t declare an `available()` method.

- Reader declares a boolean `ready()` method that returns true when the next `read()` call is guaranteed not to block for input.

- Reader declares an `int read(CharBuffer target)` method for reading characters from a character buffer. (I discuss `CharBuffer` in Chapter 13.)
OutputStreamWriter and InputStreamReader

The concrete OutputStreamWriter class (a Writer subclass) is a bridge between an incoming sequence of characters and an outgoing stream of bytes. Characters written to this writer are encoded into bytes according to the default or specified character encoding.

Note The default character encoding is accessible via the file.encoding system property.

Each call to one of OutputStreamWriter’s write() methods causes an encoder to be called on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. The characters passed to the write() methods are not buffered.

OutputStreamWriter declares four constructors, including the following pair:

- OutputStreamWriter(OutputStream out) creates a bridge between an incoming sequence of characters (passed to OutputStreamWriter via its append() and write() methods) and underlying output stream out. The default character encoding is used to encode characters into bytes.

- OutputStreamWriter(OutputStream out, String charsetName) creates a bridge between an incoming sequence of characters (passed to OutputStreamWriter via its append() and write() methods) and underlying output stream out. charsetName identifies the character encoding used to encode characters into bytes. This constructor throws java.io.UnsupportedEncodingException when the named character encoding isn't supported.

Note OutputStreamWriter depends on the abstract java.nio.charset.Charset and java.nio.charset.CharsetEncoder classes (see Chapter 13) to perform character encoding.

The following example uses the second constructor to create a bridge to an underlying file output stream so that Polish text can be written to an ISO/IEC 8859-2-encoded file:

```java
FileOutputStream fos = new FileOutputStream("polish.txt");
OutputStreamWriter osw = new OutputStreamWriter(fos, "8859_2");
char ch = '\u0323'; // Accented N.
osw.write(ch);
```

The concrete InputStreamReader class (a Reader subclass) is a bridge between an incoming stream of bytes and an outgoing sequence of characters. Characters read from this reader are decoded from bytes according to the default or specified character encoding.

Each call to one of InputStreamReader’s read() methods may cause one or more bytes to be read from the underlying input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.
InputStreamReader declares four constructors, including the following pair:

- `InputStreamReader(InputStream in)` creates a bridge between underlying input stream `in` and an outgoing sequence of characters (returned from `InputStreamReader` via its `read()` methods). The default character encoding is used to decode bytes into characters.

- `InputStreamReader(InputStream in, String charsetName)` creates a bridge between underlying input stream `in` and an outgoing sequence of characters (returned from `InputStreamReader` via its `read()` methods). `charsetName` identifies the character encoding used to decode bytes into characters. This constructor throws `UnsupportedEncodingException` when the named character encoding is not supported.

**Note** InputStreamReader depends on the abstract `Charset` and `java.nio.charset.CharsetDecoder` classes (see Chapter 13) to perform character decoding.

The following example uses the second constructor to create a bridge to an underlying file input stream so that Polish text can be read from an ISO/IEC 8859-2-encoded file:

```java
FileInputStream fis = new FileInputStream("polish.txt");
InputStreamReader isr = new InputStreamReader(fis, "8859_2");
char ch = isr.read(ch);
```

**Note** `OutputStreamWriter` and `InputStreamReader` declare a `String getEncoding()` method that returns the name of the character encoding in use. If the encoding has a historical name, that name is returned; otherwise, the encoding's canonical name is returned.

### FileWriter and FileReader

FileWriter is a convenience class for writing characters to files. It subclasses `OutputStreamWriter`, and its constructors call `OutputStreamWriter(OutputStream)`. An instance of this class is equivalent to the following code fragment:

```java
FileOutputStream fos = new FileOutputStream(pathname);
OutputStreamWriter osw;
osw = new OutputStreamWriter(fos, System.getProperty("file.encoding"));
```

In Chapter 5, I presented a logging library with a `File` class that didn't incorporate file-writing code. Listing 11-24 addresses this situation by presenting a revised `File` class that uses `FileWriter` to log messages to a file.
Listing 11-24. Logging Messages to an Actual File

```java
package logging;

import java.io.FileWriter;
import java.io.IOException;

class File implements Logger {

    private final static String LINE_SEPARATOR = System.getProperty("line.separator");

    private String dstName;
    private FileWriter fw;

    File(String dstName) {
        this.dstName = dstName;
    }

    public boolean connect() {
        if (dstName == null)
            return false;
        try {
            fw = new FileWriter(dstName);
        }
        catch (IOException ioe) {
            return false;
        }
        return true;
    }

    public boolean disconnect() {
        if (fw == null)
            return false;
        try {
            fw.close();
        }
        catch (IOException ioe) {
            return false;
        }
        return true;
    }

    public boolean log(String msg) {
        if (fw == null)
            return false;
        try {
            fw.write(msg + LINE_SEPARATOR);
        }
        catch (IOException ioe) {
            return false;
        }
        return true;
    }
}
```

www.it-ebooks.info
Listing 11-24 refactors Chapter 5’s File class to support FileWriter by making changes to each of the connect(), disconnect(), and log() methods:

- connect() attempts to instantiate FileWriter, whose instance is saved in fw on success; otherwise, fw continues to store its default null reference.
- disconnect() attempts to close the file by calling FileWriter’s close() method, but only when fw doesn’t contain its default null reference.
- log() attempts to write its String argument to the file by calling FileWriter’s void write(String str) method, but only when fw doesn’t contain its default null reference.

connect()’s catch block specifies IOException instead of FileNotFoundException because FileWriter’s constructors throw IOException when they cannot connect to existing normal files; FileOutputStream’s constructors throw FileNotFoundException.

log()’s write(String) method appends the line.separator value (which I assigned to a constant for convenience) to the string being output instead of appending \n, which would violate portability.

FileReader is a convenience class for reading characters from files. It subclasses InputStreamReader, and its constructors call InputStreamReader(InputStream). An instance of this class is equivalent to the following code fragment:

```java
FileInputStream fis = new FileInputStream(pathname);
InputStreamReader isr;
isr = new InputStreamReader(fis, System.getProperty("file.encoding"));
```

It’s often necessary to search text files for occurrences of specific strings. Although regular expressions (discussed in Chapter 13) are ideal for this task, I have yet to discuss them. As a result, Listing 11-25 presents the more verbose alternative to regular expressions.

```
Listing 11-25. Finding All Files That Contain Content Matching a Search String

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
```
public class FindAll {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java FindAll start search-string");
            return;
        }
        if (!findAll(new File(args[0]), args[1]))
            System.err.println("not a directory");
    }

    static boolean findAll(File file, String srchText) {
        File[] files = file.listFiles();
        if (files == null)
            return false;
        for (int i = 0; i < files.length; i++) {
            if (files[i].isDirectory())
                findAll(files[i], srchText);
            else
                if (find(files[i].getPath(), srchText))
                    System.out.println(files[i].getPath());
            return true;
        }
    }

    static boolean find(String filename, String srchText) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader(filename));
            int ch;
            outer_loop:
            do {
                if ((ch = br.read()) == -1)
                    return false;
                if (ch == srchText.charAt(0)) {
                    for (int i = 1; i < srchText.length(); i++) {
                        if ((ch = br.read()) == -1)
                            return false;
                        if (ch != srchText.charAt(i))
                            continue outer_loop;
                    }
                }
            } return true;
        }
    }
}
while (true);
}
catch (IOException ioe)
{
    System.err.println("I/O error: " + ioe.getMessage());
}
finally
{
    if (br != null)
        try
        {
            br.close();
        }
        catch (IOException ioe)
        {
            assert false; // shouldn't happen in this context
        }
    return false;
}
}

Listing 11-25’s FindAll class declares main(), findAll(), and find() class methods.

main() validates the number of command-line arguments, which must be two. The first argument identifies the starting location within the filesystem for the search, and is used to construct a File object. The second argument specifies search text. main() then passes the File object and the search text to findAll() to perform a search for all files containing this text.

The recursive findAll() method first invokes listFiles() on the File object passed to this method to obtain the names of all files in the current directory. If listFiles() returns null, meaning that the File object doesn’t refer to an existing directory, findAll() returns false and a suitable error message is output.

For each name in the returned list, findAll() either recursively invokes itself when the name represents a directory, or invokes the find() method to search the file for the text; the file’s pathname string is output when the file contains this text.

The find() method first opens the file identified by its first argument via the FileReader class and then passes the FileReader instance to a BufferedReader instance to improve file-reading performance. It then enters a loop that continues to read characters from the file until the end of the file is reached.

If the currently read character matches the first character in the search text, an inner loop is entered to read subsequent characters from the file and compare them with subsequent characters in the search text. When all characters match, find() returns true. Otherwise, the labeled continue statement is used to skip the remaining iterations of the inner loop and transfer execution to the labeled outer loop. After the last character has been read and there’s still no match, find() returns false.

Now that you know how FindAll works, you’ll probably want to try it out. The following examples show you how I might use this application on my Windows 7 platform:

java FindAll \prj\dev RenderScript

www.it-ebooks.info
This example searches the `\prj\dev` directory on my default drive (C:) for all files that contain the word RenderScript (case is significant) and generates the following output:

`\prj\dev\ar2\appb\ar\1-4302-4614-5_Friesen_AppB_Android_Tools_Overview.doc`
`\prj\dev\ar2\appb\ce\1-4302-4614-5_Friesen_AppB_Android_Tools_Overview.doc`
`\prj\dev\ar2\ch08\ar\1-4302-4614-5_Friesen_Ch08_Working_with_Android_NDK_and_Renderscript.doc`
`\prj\dev\ar2\ch08\ce\1-4302-4614-5_Friesen_Ch08_Working_with_Android_NDK_and_Renderscript.doc`
`\prj\dev\ar2\ch08\code\GrayScale\GrayScale.java`
`\prj\dev\ar2\ch08\code\WavyImage\WavyImage.java`
`\prj\dev\ar2\code\ch08\GrayScale\GrayScale.java`
`\prj\dev\ar2\code\ch08\WavyImage\WavyImage.java`
`\prj\dev\ar2\xtra\ndkrs.txt`
`\prj\dev\EmbossImage\src\ca\tutortutor\embossimage\EmbossImage.java`
`\prj\dev\GrayScale\src\ca\tutortutor\grayscale\GrayScale.java`
`\prj\dev\WavyImage\src\ca\tutortutor\wavyimage\WavyImage.java`

If I now specify `java FindAll \prj\dev "Jelly Bean"`, I observe the following abbreviated output:

`\prj\dev\ar2\ch01\ar\1-4302-4614-5_Friesen_Ch01_Getting_Started_with_Android.doc`
`\prj\dev\ar2\ch01\ce\1-4302-4614-5_Friesen_Ch01_Getting_Started_with_Android.doc`

**EXERCISES**

The following exercises are designed to test your understanding of Chapter 11’s content.

1. What is the purpose of the `File` class?
2. What do instances of the `File` class contain?
3. What does `File`'s `listRoots()` method accomplish?
4. What is a path and what is a pathname?
5. What is the difference between an absolute pathname and a relative pathname?
6. How do you obtain the current user (also known as working) directory?
7. Define parent pathname.
8. `File`'s constructors normalize their pathname arguments. What does normalize mean?
9. How do you obtain the default name-separator character?
10. What is a canonical pathname?
11. What is the difference between `File`'s `getParent()` and `getName()` methods?
12. True or false: `File`'s `exists()` method only determines whether or not a file exists.
13. What is a normal file?
14. What does `File`'s `lastModified()` method return?
15. True or false: File’s list() method returns an array of Strings where each entry is a filename rather than a complete path.

16. What is the difference between the FilenameFilter and FileFilter interfaces?

17. True or false: File’s createNewFile() method doesn’t check for file existence and create the file when it doesn’t exist in a single operation that’s atomic with respect to all other filesystem activities that might affect the file.

18. File’s createTempFile(String, String) method creates a temporary file in the default temporary directory. How can you locate this directory?

19. Temporary files should be removed when no longer needed after an application exits (to avoid cluttering the filesystem). How do you ensure that a temporary file is removed when the virtual machine ends normally (it doesn’t crash and the power isn’t lost)?

20. How would you accurately compare two File objects?

21. What is the purpose of the RandomAccessFile class?

22. What is the purpose of the "rwd" and "rws" mode arguments?

23. What is a file pointer?

24. True or false: When you call RandomAccessFile’s seek(long) method to set the file pointer’s value, and when this value is greater than the length of the file, the file’s length changes.

25. Define flat file database.

26. What is a stream?

27. What is the purpose of OutputStream’s flush() method?

28. True or false: OutputStream’s close() method automatically flushes the output stream.

29. What is the purpose of InputStream’s mark(int) and reset() methods?

30. How would you access a copy of a ByteArrayOutputStream instance’s internal byte array?

31. True or false: FileOutputStream and FileInputStream provide internal buffers to improve the performance of write and read operations.

32. Why would you use PipedOutputStream and PipedInputStream?

33. Define filter stream.

34. What does it mean for two streams to be chained together?

35. How do you improve the performance of a file output stream or a file input stream?

36. How do DataOutputStream and DataInputStream support FileOutputStream and FileInputStream?

37. What is object serialization and deserialization?

38. What three forms of serialization and deserialization does Java support?

39. What is the purpose of the Serializable interface?
40. What does the serialization mechanism do when it encounters an object whose class doesn’t implement `Serializable`?

41. Identify the three stated reasons for Java not supporting unlimited serialization.

42. How do you initiate serialization? How do you initiate deserialization?

43. True or false: Class fields are automatically serialized.

44. What is the purpose of the `transient` reserved word?

45. What does the deserialization mechanism do when it attempts to deserialize an object whose class has changed?

46. How does the deserialization mechanism detect that a serialized object’s class has changed?

47. How can you add an instance field to a class and avoid trouble when deserializing an object that was serialized before the instance field was added? What JDK tool can you use to help with this task?

48. How do you customize the default serialization and deserialization mechanisms without using externalization?

49. How do you tell the serialization and deserialization mechanisms to serialize or deserialize the object’s normal state before serializing or deserializing additional data items?

50. How does externalization differ from default and custom serialization and deserialization?

51. How does a class indicate that it supports externalization?

52. True or false: During externalization, the deserialization mechanism throws `InvalidClassException` with a “no valid constructor” message when it doesn’t detect a public noargument constructor.

53. What is the difference between `PrintStream's print()` and `println()` methods?

54. What does `PrintStream's noargument void println()` method accomplish?

55. Why are Java’s stream classes not good at streaming characters?

56. What does Java provide as the preferred alternative to stream classes when it comes to character I/O?

57. True or false: `Reader` declares an `available()` method.

58. What is the purpose of the `OutputStreamWriter` class? What is the purpose of the `InputStreamReader` class?

59. How do you identify the default character encoding?

60. What is the purpose of the `FileWriter` class? What is the purpose of the `FileReader` class?

61. Create a Java application named `Touch` for setting a file’s or directory’s timestamp to the current time. This application has the following usage syntax: `java Touch pathname`.

62. Improve Listing 11-11’s `Copy` application (performance wise) by using `BufferedInputStream` and `BufferedOutputStream`. `Copy` should read the bytes to be copied from the buffered input stream and write these bytes to the buffered output stream.
63. Create a Java application named Split for splitting a large file into a number of smaller part\textsubscript{x} files (where \textit{x} starts at 0 and increments; for example, part0, part1, part2, and so on). Each part\textsubscript{x} file (except possibly the last part\textsubscript{x} file, which holds the remaining bytes) will have the same size. This application has the following usage syntax: \texttt{java Split pathname}. Furthermore, your implementation must use the BufferedInputStream, BufferedOutputStream, File, FileInputStream, and FileOutputStream classes.

64. It's often convenient to read lines of text from standard input, and the InputStreamReader and BufferedReader classes make this task possible. Create a Java application named CircleInfo that, after obtaining a BufferedReader instance that is chained to standard input, presents a loop that prompts the user to enter a radius, parses the entered radius into a double value, and outputs a pair of messages that report the circle's circumference and area based on this radius.

Summary

Applications often input data for processing and output processing results. Data is input from a file or some other source and is output to a file or some other destination. Java supports I/O via the classic I/O APIs located in the \texttt{java.io} package.

File I/O activities often interact with a filesystem. Java offers access to the underlying platform's available filesystem(s) via its concrete \texttt{File} class. File instances contain the pathnames of files and directories that may or may not exist in their filesystems.

Files can be opened for random access in which a mixture of write and read operations can occur until the file is closed. Java supports this random access by providing the concrete RandomAccessFile class.

Java uses streams to perform I/O operations. A stream is an ordered sequence of bytes of arbitrary length. Bytes flow over an output stream from an application to a destination and flow over an input stream from a source to an application.

The \texttt{java.io} package provides several output stream and input stream classes that are descendents of the abstract OutputStream and InputStream classes. BufferedOutputStream and FileInputStream are examples.

Java's stream classes are good for streaming sequences of bytes but are not good for streaming sequences of characters because bytes and characters are two different things, and because byte streams have no knowledge of character sets and encodings.

If you need to stream characters, you should take advantage of Java's writer and reader classes, which were designed to support character I/O (they work with \texttt{char} instead of byte). Furthermore, the writer and reader classes take character encodings into account.

The \texttt{java.io} package provides several writer and reader classes that are descendents of the abstract Writer and Reader classes. FileWriter and FileReader are examples. These convenience classes are based on file output/input streams and OutputStreamWriter/InputStreamReader.

This chapter focused on I/O in the context of a filesystem. However, you can also perform I/O in the context of a network. Chapter 12 introduces you to several of Java's network-oriented APIs.
Accessing Networks

Applications often need to access networks to acquire resources (such as images) or to communicate with remote executable entities (such as web services). A network is a group of interconnected nodes (computing devices such as tablets, and peripherals such as scanners or laser printers) that can be shared among the network’s users.

**Note**  An intranet is a network located within an organization and an internet is a network connecting organizations to each other. The Internet is the global network of networks.

Intranets and internets often use TCP/IP (http://en.wikipedia.org/wiki/TCP/IP_model) to communicate between nodes. TCP/IP includes Transmission Control Protocol (TCP), which is a connection-oriented protocol; User Datagram Protocol (UDP), which is a connectionless protocol; and Internet Protocol (IP), which is the basic protocol over which TCP and UDP perform their tasks.

The java.net package provides types that support TCP/IP between processes (executing applications) running on the same or different hosts (computer-based TCP/IP nodes). In this chapter, I first present the types for performing socket-based and URL-based communication. I then present the low-level network interface and interface address types and cookie-oriented types.

**Note**  Android apps must have permission to access the network. Permission can be obtained by including <uses-permission android:name="android.permission.INTERNET" /> in the app manifest file.

Network-oriented applications often have to deal with the topic of endianness (http://en.wikipedia.org/wiki/Endianness), which refers to the ordering of individually addressable sub-components within the representation of a larger data item. For example, given a 16-bit short integer, do you first transmit the most significant byte or the least significant byte?
Accessing Networks via Sockets

Two processes communicate by way of sockets, which are endpoints in a communications link between these processes. Each endpoint is identified by an IP address that identifies the host and by a port number that identifies the process running on that host.

### IP Addresses and Port Numbers

An IP address is a 32-bit or 128-bit unsigned integer that uniquely identifies a network host or some other network node (a router, for example).

It is common to specify a 32-bit IP address as four 8-bit integer components in a period-separated decimal notation, where each component is a decimal integer ranging from 0 through 255 and is separated from the next component via a period (such as 127.0.0.1). A 32-bit IP address is often referred to as an Internet Protocol Version 4 (IPv4) address (see [http://en.wikipedia.org/wiki/IPv4](http://en.wikipedia.org/wiki/IPv4)).

It's common to specify a 128-bit IP address as eight 16-bit integer components in colon-separated hexadecimal notation, where each component is a hexadecimal integer ranging from 0 through FFFF and is separated from the next component via a colon (such as 1080:0:0:8:800:200C:417A). A 128-bit IP address is often referred to as an Internet Protocol Version 6 (IPv6) address (see [http://en.wikipedia.org/wiki/IPv6](http://en.wikipedia.org/wiki/IPv6)).

A port number is a 16-bit integer that uniquely identifies a process, which is the ultimate source or recipient of a message. Port numbers that are less than 1024 are reserved for standard processes. For example, port number 25 has traditionally identified the Simple Mail Transfer Protocol (SMTP) process for sending e-mail, although port number 587 has largely obsoleted this older port number (see [http://en.wikipedia.org/wiki/Smtp](http://en.wikipedia.org/wiki/Smtp)).

One process writes a message (a sequence of bytes) to its socket. The network management software portion of the underlying platform breaks the message into a sequence of packets (addressable message chunks that are often referred to as IP datagrams), and forwards them to the other process’s socket where they are recombined into the original message for processing.

Figure 12-1 shows how two sockets communicate in a TCP/IP context.
In the context of Figure 12-1, suppose that Process A wants to send a message to Process B. Process A sends that message to its socket with the destination socket address of Process B. Host A's network management software (often referred to as a protocol stack) obtains this message and reduces it to a sequence of packets, with each packet including the destination host's IP address and port number. The network management software then sends these packets through Host A's Network Interface Card (NIC) to Host B.

Host B's protocol stack receives packets through the NIC and reassembles them into the original message (packets may be received out of order), which it then makes available to Process B via its socket. This scenario reverses when Process B communicates with Process A.

The network management software uses TCP to create an ongoing conversation between two hosts in which messages are sent back and forth. Before this conversation occurs, a connection is established between these hosts. After the connection has been established, TCP enters a pattern where it sends message packets and waits for a reply that they arrived correctly (or for a timeout to expire when the reply doesn't arrive because of some network problem). This pattern repeats and guarantees a reliable connection. For detailed information on this pattern, check out http://en.wikipedia.org/wiki/Tcp_receive_window#Flow_control.

**Note** The NIC's various network interfaces are connections between a computer and a network.
Because it can take time to establish a connection, and it also takes time to send packets (as it is necessary to receive reply acknowledgments and also because of timeouts), TCP is slow. On the other hand, UDP, which doesn’t require connections and packet acknowledgement, is much faster. The downside is that UDP isn’t as reliable (there’s no guarantee of packet delivery, ordering, or protection against duplicate packets, although UDP uses checksums to verify that data is correct) because there’s no acknowledgment. Furthermore, UDP is limited to single-packet conversations.

The java.net package provides Socket, ServerSocket, and other Socket-suffixed classes for performing TCP-based or UDP-based communications. Before investigating these classes, you need to understand socket addresses and socket options.

**Socket Addresses**

An instance of a Socket-suffixed class is associated with a socket address comprised of an IP address and a port number. These classes often rely on the InetAddress class to represent the IPv4 or IPv6 address portion of the socket address, and represent the port number separately.

> **Note**  InetAddress relies on its Inet4Address subclass to represent an IPv4 address and on its Inet6Address subclass to represent an IPv6 address.

InetAddress declares several class methods for obtaining an InetAddress instance. These methods include the following:

- InetAddress[] getAllByName(String host) returns an array of InetAddresses that store the IP addresses associated with host. You can pass either a domain name (such as tutortutor.ca) or an IP address (such as 70.33.247.10) argument to this parameter. (To learn about domain names, check out Wikipedia's “Domain name” entry at http://en.wikipedia.org/wiki/Domain_name.) Pass null to obtain an InetAddress instance that stores the IP address of the loopback interface (a software-based network interface where outgoing data loops back as incoming data). This method throws UnknownHostException when no IP address for the specified host can be found, or when a scope identifier is specified for a global IPv6 address.

- InetAddress getAddress(byte[] addr) returns an InetAddress object for the given raw IP address. The argument passed to addr is in network byte order (most significant byte comes first) where the highest order byte is stored in addr[0]. The addr array’s length must be 4 bytes for an IPv4 address and 16 bytes for an IPv6 address. This method throws UnknownHostException when the array has another length.

- InetAddress getAddress(String hostName, byte[] ipAddress) returns an InetAddress instance based on the host name and IP address arguments. This method throws UnknownHostException when the array’s length is neither 4 nor 16.
InetAddress getByName(String host) returns an InetAddress instance based on the host argument, which can be a machine name (such as tutor.tutor.ca) or a textual representation of its IP address. Passing null to host results in an InetAddress instance representing an address of the loopback interface being returned.

InetAddress getLocalHost() returns the address of the local host (the current host), which is represented by hostname localhost or by an IP address that's commonly expressed as 127.0.0.1 (IPv4) or ::1 (IPv6). This method throws UnknownHostException when local host couldn't be resolved into an address.

After you obtain an InetAddress instance, you can interrogate it by invoking instance methods such as byte[] getAddress(), which returns the raw IP address (in network byte order) of this InetAddress object, and boolean isLoopbackAddress(), which determines whether or not this InetAddress instance represents a loopback address.

Java 1.4 introduced the abstract SocketAddress class to represent a socket address “with no protocol attachment.” (This class's creator might have anticipated that Java would eventually support low-level communication protocols other than the widely popular Internet Protocol.)

SocketAddress is subclassed by the concrete InetSocketAddress class, which represents a socket address as an IP address and a port number. It can also represent a hostname and a port number and will make an attempt to resolve the hostname.

InetSocketAddress instances are created by invoking InetSocketAddress(InetAddress addr, int port) and other constructors. After an instance has been created, you can call methods such as InetAddress getAddress() and int getPort() to return socket address components.

Socket Options

An instance of a Socket-suffixed class shares the concept of socket options, which are parameters for configuring socket behavior. Socket options are described by constants that are declared in the SocketOptions interface.

- IP_MULTICAST_IF specifies the outgoing network interface for multicast packets (on multihomed [multiple NIC] hosts). This option isn't implemented by Android.
- IP_MULTICAST_IF2 specifies the outgoing network interface for multicast packets using an interface index.
- IP_MULTICAST_LOOP enables or disables local loopback of multicast datagrams.
- IP_TOS sets the type-of-service (IPv4) or traffic class (IPv6) field in the IP header for a TCP or UDP socket.
- SO_BINDADDR fetches the socket’s local address binding. This option isn’t implemented by Android.
- SO_BROADCAST enables a socket to send broadcast messages.
- SO_KEEPALIVE turns on socket keepalive.
- SO_LINGER specifies the number of seconds to wait when closing a socket when there is still some buffered data to be sent.
SO_OOBINLINE enables inline reception of TCP urgent data.

SO_RCVBUF sets or gets the maximum socket receive buffer size (in bytes).

SO_REUSEADDR enables a socket's reuse address.

SO_SNDBUF sets or gets the maximum socket send buffer size (in bytes).

SO_TIMEOUT specifies a timeout (in milliseconds) on blocking accept or read/receive (but not write/send) socket operations. (Don’t block forever!)

TCP_NODELAY disables Nagle’s algorithm (http://en.wikipedia.org/wiki/Nagle's_algorithm). Written data to the network is not buffered pending acknowledgement of previously written data.

SocketOptions also declares the following methods for setting and getting these options:

- void setOption(int optID, Object value)
- Object getOption(int optID)

optID is one of the aforementioned constants and value is an object of a suitable class (such as java.lang.Boolean).

SocketOptions is implemented by the abstract SocketImpl and DatagramSocketImpl classes. Concrete instances of these classes are wrapped by the various Socket-suffixed classes. As a result, you cannot invoke these methods. Instead, you work with the type-safe setter and getter methods provided by the Socket-suffixed classes for setting and getting these options.

For example, Socket declares void setKeepAlive(boolean keepAlive) for setting the SO_KEEPALIVE option, and ServerSocket declares void setSoTimeout(int timeout) for setting the SO_TIMEOUT option. Check the documentation on the Socket-suffixed classes to learn about these and other socket option methods.

Note  Socket option methods that apply to DatagramSocket also apply to its MulticastSocket subclass.

Socket and ServerSocket

The Socket and ServerSocket classes support TCP-based communications between client processes (such as an application running on a tablet) and server processes (such as an application running on one of your Internet Service Provider's computers that provides access to the World Wide Web). Because Socket is associated with the java.io.InputStream and java.io.OutputStream classes, sockets based on the Socket class are commonly referred to as stream sockets.

Socket supports the creation of client-side sockets. It declares several constructors for this purpose, including the following pair:

- Socket(InetAddress dstAddress, int dstPort) creates a stream socket and connects it to the specified port number (described by dstPort) at the specified IP address (described by dstAddress). This constructor throws java.io.IOException when an I/O error occurs while creating the socket, java.lang. Exception when an error occurs.
IllegalArgumentException when the argument passed to dstPort is outside the valid range of port values, which is 0 through 65535, and java.lang.
NullPointerException when dstAddress is null.

Socket(String dstName, int dstPort) creates a stream socket and connects it to the port identified by dstPort on the host identified by dstName. When dstName is null, this constructor is equivalent to invoking Socket(InetAddress. getByName(null), port). It throws the same IOException and IllegalArgumentException instances as the previous constructor. However, instead of throwing NullPointerException, it throws UnknownHostException when the host's IP address cannot be determined.

After a Socket instance is created via these constructors, it's bound to an arbitrary local host socket address before a connection is made to the remote host socket address. Binding makes a client socket address available to a server socket so that a server process can communicate with the client process via the server socket.

Socket offers additional constructors. For example, Socket() and Socket(Proxy proxy) create unbound and unconnected sockets. Before using these sockets, they must be bound to local socket addresses, by calling void bind(SocketAddress localAddr), and then connections must be made, by calling Socket's connect() methods (void connect(SocketAddress remoteAddr), for example).

Note A proxy is a host that sits between an intranet and the Internet for security purposes. Proxy settings are represented via instances of the Proxy class and help sockets communicate through proxies.

Another constructor is Socket(InetAddress dstAddress, int dstPort, InetAddress localAddr, int localPort), which lets you specify your own local host socket address via localAddr and localPort. This constructor automatically binds to the local socket address and then attempts a connection to the remote dstPort on dstAddress.

After creating a Socket instance, and possibly invoking bind() and connect() on that instance, an application invokes Socket's InputStream getInputStream() and OutputStream getOutputStream() methods to acquire an input stream for reading bytes from the socket and an output stream for writing bytes to the socket. Also, the application often calls Socket's void close() method to close the socket when no longer needed for I/O.

The following example demonstrates how to create a socket that's bound to port number 9999 on the local host and then access its input and output streams—exceptions are ignored for brevity:

    Socket socket = new Socket("localhost", 9999);
    InputStream is = socket.getInputStream();
    OutputStream os = socket.getOutputStream();
    // Do some work with the socket.
    socket.close();
ServerSocket supports the creation of server-side sockets. It declares the following four constructors for this purpose:

- **ServerSocket()** creates an unbound server socket. You can bind this socket to a specific socket address (to which client sockets communicate) by invoking either of ServerSocket's two `bind()` methods. Binding makes the server socket address available to a client socket so that a client process can communicate with the server process via the client socket. This constructor throws IOException when an I/O error occurs while attempting to open the socket.

- **ServerSocket(int port)** creates a server socket bound to the specified port value and an IP address associated with one of the host's NICs. When you pass 0 to `port`, an arbitrary port number is chosen. The port number can be retrieved by calling `getLocalPort()`. The maximum queue length for incoming connection requests from clients is set to 50. If a connection request arrives when the queue is full, the connection is refused. This constructor throws IOException when an I/O error occurs while attempting to open the socket and IllegalArgumentException when `port`'s value lies outside the specified range of valid port values, which is between 0 and 65535, inclusive.

- **ServerSocket(int port, int backlog)** is equivalent to the previous constructor, but it also lets you specify the maximum queue length for incoming connections by passing a positive integer to `backlog`.

- **ServerSocket(int port, int backlog, InetAddress localAddress)** is equivalent to the previous constructor, but it also lets you specify a different IP address to which the server socket binds. (Any address is chosen when null is passed.) This constructor is useful for machines that have multiple NICs and you want to listen for connection requests on a specific NIC.

After a server socket is created via these constructors, a server application enters a loop that first invokes `ServerSocket`'s `Socket accept()` method to listen for a connection request and return a `Socket` instance that lets it communicate with the associated client socket. It then communicates with the client socket to perform some kind of processing. When processing finishes, the server socket calls the client socket's `close()` method to terminate its connection with the client.

**Note** ServerSocket declares a void `close()` method for closing a server socket before terminating the server application. An unclosed socket is automatically closed when an application terminates.

The following example demonstrates how to create a server socket that's bound to port 9999 on the current host, listen for incoming connection requests, return their sockets, perform work on those sockets, and close the sockets—exceptions are ignored for brevity:

```java
ServerSocket ss = new ServerSocket(9999);
while (true)
{
    Socket socket = ss.accept();
```
The `accept()` method call blocks until a connection request is available and then returns a `Socket` object so that the server application can communicate with its associated client. The socket is closed after this communication takes place. The server socket is automatically closed when the application exits.

This example assumes that socket communication takes place on the server application's main thread, which is a problem when processing takes time to perform because server response time to incoming connection requests decreases. To speed up response time, it's often necessary to communicate with the socket on a worker thread, as demonstrated in the following example:

```java
ServerSocket ss = new ServerSocket(9999);
while (true)
{
    final Socket s = ss.accept();
    new Thread(new Runnable()
    {
        @Override
        public void run()
        {
            // obtain socket input/output streams and communicate with socket
            try { s.close(); } catch (IOException ioe) {} 
        }
    }).start();
}
```

Each time a connection request arrives, `accept()` returns a `Socket` instance, and then a `java.lang.Thread` object is created whose Runnable accesses that socket for communicating with the socket on a worker thread.

**Tip** Although this example uses the `Thread` class, you could use an executor (see Chapter 10) instead.

I've created `EchoClient` and `EchoServer` applications that demonstrate `Socket` and `ServerSocket`. Listing 12-1 presents `EchoClient`'s source code.

**Listing 12-1. Echoing Data to and Receiving It Back from a Server**

```java
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
```
import java.net.Socket;
import java.net.UnknownHostException;

public class EchoClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage : java EchoClient message");
            System.err.println("example: java EchoClient "This is a test."");
            return;
        }
        try {
            Socket socket = new Socket("localhost", 9999);
            OutputStream os = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(os);
            PrintWriter pw = new PrintWriter(osw);
            pw.println(args[0]);
            pw.flush();
            InputStream is = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            System.out.println(br.readLine());
        } catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + uhe.getMessage());
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}

EchoClient first verifies that it has received a single command-line argument and then creates a
socket that will connect to a process running on port 9999 of the local host.

After creating the socket, EchoClient obtains an output stream for writing a string to the socket.
Because the output stream can only handle a sequence of bytes, the java.io.OutputStreamWriter
and java.io.PrintWriter classes (see Chapter 11) are used to connect the writer that outputs
characters to the byte-oriented output stream.

After instantiating PrintWriter, EchoClient invokes its void println(String str) method to write
the string followed by a newline character. The void flush() method is subsequently called to
ensure that all pending data is written to the server.

EchoClient now obtains an input stream for reading the string as a sequence of bytes. It then
connects the reader (that inputs characters) to the byte-oriented input stream by instantiating
java.io.InputStreamReader and java.io.BufferedReader (see Chapter 11).
Finally, EchoClient invokes BufferedReader’s String \texttt{readLine()} method to read the characters followed by a newline from the socket. \texttt{(readLine()) doesn’t include the newline character in the returned string.) These characters followed by a newline are then written to standard output.

\textbf{Note}  In a long-running application, you would explicitly close the socket instance by invoking its \texttt{void close()} method when the socket is no longer needed. For brevity, I’ve chosen not to do so in this and most of the remaining Socket-suffixed class examples.

Listing 12-2 presents EchoServer’s source code.

\textit{Listing 12-2. Receiving Data from and Echoing It Back to a Client}

\begin{verbatim}
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        System.out.println("Starting echo server...");
        ServerSocket ss = new ServerSocket(9999);
        while (true) {
            Socket s = ss.accept();
            try {
                InputStream is = s.getInputStream();
                InputStreamReader isr = new InputStreamReader(is);
                BufferedReader br = new BufferedReader(isr);
                String msg = br.readLine();
                System.out.println(msg);
                OutputStream os = s.getOutputStream();
                OutputStreamWriter osw = new OutputStreamWriter(os);
                PrintWriter pw = new PrintWriter(osw);
                pw.println(msg);
                pw.flush();
            }
        }
    }
}
\end{verbatim}
EchoServer first outputs an introductory message to standard output and then creates a server socket that listens for connections on port 9999. It then enters an infinite loop, where each iteration invokes ServerSocket's Socket accept() method to block until a connection is received and then returns a Socket object representing this connection.

After obtaining the socket, EchoServer obtains an input stream for reading from the socket. Because the input stream can only handle a sequence of bytes, the InputStreamReader and BufferedReader classes are used to connect the reader that inputs characters to the byte-oriented input stream.

EchoServer now obtains an output stream for writing the string as a sequence of bytes. It then connects the writer that outputs characters to the byte-oriented output stream by instantiating OutputStreamWriter and PrintWriter.

After outputting the message to standard output, EchoServer calls flush() to flush the output to the client. The client socket is then closed.

To experiment with these applications, copy EchoClient.java and EchoServer.java to the same directory and open two console windows with this directory being current. Compile each source file and execute java EchoServer in one window—you should observe an introductory message, although you might first need to enable port 9999 on the firewall (http://en.wikipedia.org/wiki/Firewall_(computing)). Having started the server, echo the following command to echo text to both windows:

```
java EchoClient "This is a test."
```

You should observe “This is a test.” in both windows.

### DatagramSocket and MulticastSocket

The DatagramSocket and MulticastSocket classes let you perform UDP-based communications between a pair of hosts (DatagramSocket) or between many hosts (MulticastSocket). With either class, you communicate one-way messages via datagram packets, which are arrays of bytes associated with instances of the DatagramPacket class.
Note Although you might think that Socket and ServerSocket are all that you need, DatagramSocket and its MulticastSocket subclass have their uses. For example, consider a scenario in which a group of machines need to occasionally tell a server that they're alive. It shouldn't matter when the occasional message is lost or even when the message doesn't arrive on time. Another example is a low-priority stock ticker that periodically broadcasts stock prices. When a packet doesn't arrive, odds are that the next packet will arrive and you'll then receive notification of the latest prices. Timely rather than reliable or orderly delivery is more important in real-time applications.

DatagramPacket declares several constructors with DatagramPacket(byte[] buf, int length) being the simplest. This constructor requires you to pass byte array and integer arguments to buf and length, where buf is a data buffer that stores data to be sent or received, and length (which must be less than or equal to buf.length) specifies the number of bytes (starting at buf[0]) to send/receive.

The following example demonstrates this constructor:

```java
byte[] buffer = new byte[100];
DatagramPacket dgp = new DatagramPacket(buffer, buffer.length);
```

Note Additional constructors let you specify an offset into buf that identifies the storage location of the first outgoing or incoming byte, and/or let you specify a destination socket address.

DatagramSocket describes a socket for the client or server side of the UDP-communication link. Although this class declares several constructors, I find it convenient in this chapter to use the DatagramSocket() constructor for the client side and the DatagramSocket(int port) constructor for the server side. Either constructor throws SocketException when it cannot create the datagram socket or bind the datagram socket to a local port.

After an application instantiates DatagramSocket, it calls void send(DatagramPacket dgp) and void receive(DatagramPacket dgp) to send and receive datagram packets.

Listing 12-3 demonstrates DatagramPacket and DatagramSocket in a server context.

Listing 12-3. Receiving Datagram Packets from and Echoing Them Back to Clients

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

public class DGServer {
    final static int PORT = 10000;
    
    public static void main(String[] args) {
        DatagramSocket ds = null;
        try {
            ds = new DatagramSocket(PORT);
```
public static void main(String[] args) throws SocketException
{
System.out.println("Server is starting");
DatagramSocket dgs = new DatagramSocket(PORT);
try
{
System.out.println("Send buffer size = " + dgs.getSendBufferSize());
System.out.println("Receive buffer size = " +
dgs.getReceiveBufferSize());
byte[] data = new byte[100];
DatagramPacket dgp = new DatagramPacket(data, data.length);
while (true)
{
dgs.receive(dgp);
System.out.println(new String(data));
dgs.send(dgp);
}
}
catch (IOException ioe)
{
System.err.println("I/O error: " + ioe.getMessage());
}
}

Listing 12-3’s main() method first creates a DatagramSocket object and binds the socket to port 10000 on the local host. It then invokes DatagramSocket’s int getSendBufferSize() and int getReceiveBufferSize() methods to get the values of the SO_SNDBUF and SO_RCVBUF socket options, which are then output.

Note Sockets are associated with underlying platform send and receive buffers, and their sizes are accessed by calling getSendBufferSize() and getReceiveBufferSize(). Similarly, their sizes can be set by calling DatagramSocket’s void setReceiveBufferSize(int size) and void setSendBufferSize(int size) methods. Although you can adjust these buffer sizes to improve performance, there’s a practical limit with regard to UDP. The maximum size of a UDP packet that can be sent or received is 65,507 bytes under IPv4—it’s derived from subtracting the 8-byte UDP header and 20-byte IP header values from 65,535. Although you can specify a send/receive buffer with a greater value, doing so is wasteful because the largest packet is restricted to 65,507 bytes. Also, attempting to send or receive a packet with a buffer length that exceeds 65,507 bytes results in IOException.

main() next instantiates DatagramPacket in preparation for receiving a datagram packet from a client and then echoing the packet back to the client. It assumes that packets will be 100 bytes or less in size.

Finally, main() enters an infinite loop that receives a packet, outputs packet content, and sends the packet back to the client—the client’s addressing information is stored in DatagramPacket.
Compile Listing 12-3 (javac DGServer.java) and run the application (java DGServer). You should observe output that's the same as or similar to that shown below:

Server is starting
Send buffer size = 8192
Receive buffer size = 8192

Listing 12-4 demonstrates DatagramPacket and DatagramSocket in a client context.

Listing 12-4. Sending a Datagram Packet to and Receiving It Back from a Server

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class DGClient
{
    final static int PORT = 10000;
    final static String ADDR = "localhost";

    public static void main(String[] args) throws SocketException
    {
        System.out.println("client is starting");
        DatagramSocket dgs = new DatagramSocket();
        try
        {
            byte[] buffer;
            buffer = "Send me a datagram".getBytes();
            InetAddress ia = InetAddress.getByName(ADDR);
            DatagramPacket dgp = new DatagramPacket(buffer, buffer.length, ia, PORT);
            dgs.send(dgp);

            byte[] buffer2 = new byte[100];
            dgp = new DatagramPacket(buffer2, buffer.length, ia, PORT);
            dgs.receive(dgp);
            System.out.println(new String(dgp.getData()));
        }
        catch (IOException ioe)
        {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}

Listing 12-4 is similar to Listing 12-3, but there’s one big difference. I use the DatagramPacket(byte[], int, InetAddress, int) constructor to specify the server’s destination, which happens to be port 10000 on the local host, in the datagram packet. The send() method call routes the packet to this destination.
Compile Listing 12-4 (javac DGClient.java) and run the application (java DGClient). Assuming that DGServer is also running, you should observe the following output in DGClient's command window (and the last line of this output in DGServer's command window):

```
client is starting
Send me a datagram
```

`MulticastSocket` describes a socket for the client or server side of a UDP-based multicasting session. Two commonly used constructors are `MulticastSocket()` (it creates a multicast socket not bound to a port) and `MulticastSocket(int port)` (it creates a multicast socket bound to the specified port). Either constructor throws `IOException` when an I/O error occurs.

### WHAT IS MULTICASTING?

Previous examples have demonstrated uncasting, which occurs when a server sends a message to a single client. However, it's also possible to broadcast the same message to multiple clients (such as transmit a “school closed due to bad weather” announcement to all members of a group of parents who have registered with an online program to receive this announcement); this activity is known as multicasting.

A server multicasts by sending a sequence of datagram packets to a special IP address, which is known as a multicast group address, and a specific port (as specified by a port number). Clients wanting to receive those datagram packets create a multicast socket that uses that port number. They request to join the group through a **join group operation** that specifies the special IP address. At this point, the client can receive datagram packets sent to the group and can even send datagram packets to other group members. After the client has read all datagram packets that it wants to read, it removes itself from the group by applying a **leave group operation** that specifies the special IP address.

IPv4 addresses 224.0.0.1 to 239.255.255.255 (inclusive) are reserved for use as multicast group addresses.

Listing 12-5 presents a multicasting server.

#### Listing 12-5. Multicasting Datagram Packets

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class MCServer
{
    final static int PORT = 10000;

    public static void main(String[] args)
    {
        try
        {
            MulticastSocket mcs = new MulticastSocket();
            InetAddress group = InetAddress.getByName("231.0.0.1");
            byte[] dummy = new byte[0];
```
CHAPTER 12: Accessing Networks

Listing 12-5's main() method first creates a MulticastSocket instance via the MulticastSocket() constructor. The multicast socket doesn’t need to bind to a port number because the port number is specified along with the multicast group’s IP address (231.0.0.1) as part of the DatagramPacket instance that’s subsequently created. (The dummy array is present to prevent a NullPointerException object from being thrown from the DatagramPacket constructor—this array isn't used to store data to be broadcasted.)

At this point, main() enters an infinite loop that first creates an array of bytes from a java.lang.String instance, and uses the platform’s default character encoding (see Chapter 11) to convert from Unicode characters to bytes. (Although extraneous java.lang.StringBuilder and String objects are created via expression "line " + i in each loop iteration, I’m not worried about their impact on garbage collection in this short throwaway application.)

This data buffer is subsequently assigned to the DatagramPacket instance by calling its void setData(byte[] buf) method, and then the datagram packet is broadcast to all members of the group associated with port 10000 and multicast IP address 231.0.0.1.

Compile Listing 12-5 (javac MCServer.java) and run this application (java MCServer). You shouldn’t observe any output.

Listing 12-6 presents a multicasting client.

Listing 12-6. Receiving Multicasted Datagram Packets

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class MCClient
{
    final static int PORT = 10000;

    public static void main(String[] args) throws IOException
    {
        MulticastSocket mcs = new MulticastSocket();
        InetAddress group = InetAddress.getByName("231.0.0.1");
        DatagramPacket dummy = new DatagramPacket(new byte[0], 0);
        mcs.joinGroup(group);

        DatagramPacket dgp = new DatagramPacket(dummy, 0, group, PORT);
        int i = 0;
        while (true)
        {
            byte[] buffer = ("line " + i).getBytes();
            dgp.setData(buffer);
            dgp.setLength(buffer.length);
            mcs.send(dgp);
            i++;
        }
        catch (IOException ioe)
        {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

Compile Listing 12-6 (javac MCClient.java) and run this application (java MCClient). You should receive output when you run the server application (java MCServer).
public static void main(String[] args) {
    try {
        MulticastSocket mcs = new MulticastSocket(PORT);
        InetAddress group = InetAddress.getByName("231.0.0.1");
        mcs.joinGroup(group);
        for (int i = 0; i < 10; i++) {
            byte[] buffer = new byte[256];
            DatagramPacket dgp = new DatagramPacket(buffer, buffer.length);
            mcs.receive(dgp);
            byte[] buffer2 = new byte[dgp.getLength()];
            System.arraycopy(dgp.getData(), 0, buffer2, 0, dgp.getLength());
            System.out.println(new String(buffer2));
        }
        mcs.leaveGroup(group);
    } catch (IOException ioe) {
        System.err.println("I/O error: " + ioe.getMessage());
    }
}

Listing 12-6’s main() method first creates a MulticastSocket instance bound to port 10000 via the MulticastSocket(int port) constructor.

It then obtains an InetAddress object that contains multicast group IP address 231.0.0.1 and uses this object to join the group at this address by calling MulticastSocket's void joinGroup(InetAddress mcastaddr) method.

main() next receives 10 datagram packets, prints their contents, and leaves the group by calling MulticastSocket's void leaveGroup(InetAddress mcastaddr) method with the same multicast IP address as its argument.

Note joinGroup() and leaveGroup() throw IOException when an I/O error occurs while attempting to join or leave the group or when the IP address is not a multicast IP address.

Because the client doesn’t know exactly how long the arrays of bytes will be, it assumes 256 bytes to ensure that the data buffer will hold the entire array. If it tried to print out the returned array, you would see a lot of empty space after the actual data had been printed.

To eliminate this space, the client invokes DatagramPacket's int getLength() method to obtain the actual length of the array, creates a second byte array (buffer2) with this length, and uses System.arraycopy()—discussed in Chapter 7—to copy this many bytes to buffer2. After converting this byte array to a String object (via the String(byte[] bytes) constructor, which uses the platform’s default character set—see Chapter 11 to learn about character sets), it prints the resulting characters to the standard output stream.
Compile Listing 12-6 (javac MCClient.java) and run this application (java MCClient). You should observe output similar to the following:

line 462615
line 462616
line 462617
line 462618
line 462619
line 462620
line 462621
line 462622
line 462623
line 462624

## Accessing Networks via URLs

A *Uniform Resource Locator (URL)* is a character string that specifies where a resource (such as a web page) is located on a TCP/IP-based network (such as the Internet). Also, it provides the means to retrieve that resource. For example, `http://tutortutor.ca` is a URL that locates my web site’s main page. The `http://` prefix specifies that *HyperText Transfer Protocol (HTTP)*, which is a high-level protocol on top of TCP/IP for locating HTTP resources (such as web pages), must be used to retrieve the web page located at tutortutor.ca.

### URN, URL, AND URI

A *Uniform Resource Name (URN)* is a character string that names a resource and doesn’t provide a way to access that resource (the resource might not be available). For example, `urn:isbn:9781430264545` identifies an Apress book named *Learn Java for Android Development* and that’s all.

URNs and URLs are examples of *Uniform Resource Identifiers (URIs)*, which are character strings for identifying names (URNs) and resources (URLs). Every URN and URL is also a URI.

The `java.net` package provides `URL` and `URLConnection` classes for accessing URL-based resources. It also provides `URLEncoder` and `URLDecoder` classes for encoding and decoding URLs as well as the `URI` class for performing URI-based operations (such as relativization) and returning `URL` instances containing the results.

### URL and URLConnection

The `URL` class represents URLs and provides access to the resources to which they refer. Each `URL` instance unambiguously identifies an Internet resource.

`URL` declares several constructors, with `URL(String s)` being the simplest. This constructor creates a `URL` instance from the `String` argument passed to `s` and is demonstrated as follows:

```java
try {
    URL url = new URL("http://tutortutor.ca");
}
```
catch (MalformedURLException murle) {
    // handle the exception
}

This example creates a URL object that uses HTTP to access the web page at http://tutortutor.ca. If I specified an illegal URL (such as foo), the constructor would throw MalformedURLException (an IOException subclass).

Although you'll commonly specify http:// as the protocol prefix, this isn't your only choice. For example, you can also specify file:/// when the resource is located on the local host. Furthermore, you can prepend jar: to either http:// or file:/// when the resource is stored in a JAR file, as demonstrated here:

```
jar:file:///C:/rt.jar!/java/util/Timer.class
```

The jar: prefix indicates that you want to access a JAR file resource (such as a stored classfile). The file:// prefix identifies the local host's resource location, which happens to be rt.jar (Java 5's runtime JAR file) in the current directory on the Windows C: hard drive in this example.

The path to the JAR file is followed by an exclamation mark (!) to separate the JAR file path from the JAR resource path, which happens to be the /java/util/Timer.class classfile entry in this JAR file (the leading / character is required).

---

**Note** The URL class in Oracle's Java reference implementation supports additional protocols, including ftp.

After creating a URL object, you can invoke various URL methods to access portions of the URL. For example, String getProtocol() returns the protocol portion of the URL (such as http). You can also retrieve the resource by calling the InputStream openStream() method.

```
InputStream is = url.openStream();
int ch;
while ((ch = is.read()) != -1)
    System.out.print((char) ch);
```

---

**Note** For an HTTP connection, an internal socket is created that connects to HTTP port 80 on the server identified via the URL's domain name/IP address, unless you append a different port number to the domain name/IP address (such as http://tutortutor.ca:8080).
I've created a ListResource application that demonstrates URL by using this class to fetch a resource and list its contents. Listing 12-7 presents ListResource's source code.

Listing 12-7. Listing the Contents of the Resource Identified via a URL Command-Line Argument

```java
import java.io.InputStream;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

public class ListResource
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java ListResource url");
            return;
        }
        try
        {
            URL url = new URL(args[0]);
            InputStream is = url.openStream();
            try
            {
                int ch;
                while ((ch = is.read()) != -1)
                {
                    System.out.print((char) ch);
                }
            }
            catch (IOException ioe)
            {
                is.close();
            }
        }
        catch (MalformedURLException murle)
        {
            System.err.println("invalid URL");
        }
        catch (IOException ioe)
        {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```
ListResource first verifies that it has received a single command-line argument and then attempts to instantiate URL with this argument. Assuming that the URL is valid, which means that MalformedURLException isn’t thrown, ListResource calls openStream() on the URL instance and proceeds to list the resource contents to standard output.

Compile this source code (javac ListResource.java) and execute the following command to list the contents of the page at http://tutortutor.ca:

```java
java ListResource http://tutortutor.ca
```

The following output presents a short prefix of the returned web page:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>
TutorTutor -- /main
</title>
<link rel="stylesheet" href="/shared/styles.css" media="screen">
```

openStream() is a convenience method for invoking openConnection().getInputStream(). Each of URL’s URLConnection openConnection() and URLConnection openConnection(Proxy proxy) methods returns an instance of the URLConnection class, which represents a communications link between the application and a URL.

URLConnection gives you additional control over client/server communication. For example, you can use this class to output content to various resources that accept content. In contrast, URL only lets you input content via openStream().

URLConnection declares various methods, including the following:

- **InputStream** getInputStream() returns an input stream that reads from this open connection.
- **OutputStream** getOutputStream() returns an output stream that writes to this open connection.
- void setDoInput(boolean doInput) specifies that this URLConnection object supports (pass true to doInput) or doesn’t support (pass false to doInput) input. Because true is the default, you would only pass true to this method to document your intention to perform input.
- void setDoOutput(boolean doOutput) specifies that this URLConnection object supports (pass true to doOutput) or doesn’t support (pass false to doOutput) output. Because false is the default, you must call this method before you can perform output.
- void setRequestProperty(String field, String newValue) sets a request property (such as HTTP’s accept property). When a field already exists, its value is overwritten with the specified value.
The following example shows you how to obtain an URLConnection object from the URL object referenced by the precreated url variable, enable its dooutput property, and obtain an output stream for writing to the resource:

```java
URLConnection urlc = url.openConnection();
urlc.setDoOutput(true);
OutputStream os = urlc.getOutputStream();
```

URLConnection is subclassed by HttpURLConnection and JarURLConnection. These classes declare constants and/or methods that are specific to working with the HTTP protocol or interacting with JAR-based resources.

For example, HttpURLConnection declares `void setRequestMethod(String method)` for specifying the HTTP request command to be sent to a remote HTTP server. GET and POST are commonly specified commands.

**Note** For brevity, I refer you to the Java documentation on `URLConnection`, `HttpURLConnection`, and `JarURLConnection` to learn more about these classes.

### URLEncoder and URLDecoder

HyperText Markup Language (HTML) lets you introduce forms into web pages that solicit information from page visitors. After filling out a form’s fields, the visitor clicks the form’s Submit button (which may specify something other than Submit) and the form content (field names and values) is sent to a server program. Before sending the form content, a web browser encodes this data by replacing spaces and other URL-illegal characters, and sets the content’s Internet media type (also known as Multipurpose Internet Mail Extensions [MIME] type) to `application/x-www-form-urlencoded`.

**Note** The data is encoded for HTTP POST and HTTP GET operations. Unlike POST, GET requires a *query string* (a ?-prefixed string containing the encoded content) to be appended to the server program’s URL.

The java.net package provides URLEncoder and URLDecoder classes to assist you with the tasks of encoding and decoding form content.

**URLEncoder** applies the following encoding rules:

- Alphanumeric characters (a-z, A-Z, and 0-9) remain the same.
- Special characters “.”, “-”, “*”, and “_” remain the same.
- The space character “ ” is converted into a plus sign “+”.

www.it-ebooks.info
All other characters are unsafe and are first converted into 1 or more bytes using some encoding scheme. Each byte is then represented by the three-character string %xy, where xy is the 2-digit hexadecimal representation of that byte. The recommended encoding scheme to use is UTF-8. However, for compatibility reasons, the platform’s default encoding is used when an encoding isn’t specified.

For example, using UTF-8 as the encoding scheme, the string "string ü@foo-bar" is converted to "string+%C3%BC%40foo-bar". In UTF-8, character ü is encoded as 2 bytes C3 (hex) and BC (hex); and character @ is encoded as 1 byte 40 (hex).

URLEncoder declares the following class method for encoding a string:

```java
String encode(String s, String enc)
```

This method translates the String argument passed to s into application/x-www-form-urlencoded format using the encoding scheme specified by enc. It uses the supplied encoding scheme to obtain the bytes for unsafe characters, and throws java.io.UnsupportedEncodingException when enc’s value isn’t supported.

URLDecoder applies the following decoding rules:

- Alphanumeric characters (a-z, A-Z, and 0-9) remain the same.
- Special characters “.”, “_”, “*”, and “_” remain the same.
- The plus sign “+” is converted into a space character “ ”.
- A sequence of the form %xy will be treated as representing a byte, where xy is the 2-digit hexadecimal representation of the 8 bits. Then, all substrings containing one or more of these byte sequences consecutively will be replaced by the character(s) whose encoding would result in those consecutive bytes. The encoding scheme used to decode these characters may be specified; when unspecified, the platform’s default encoding is used.

URLDecoder declares the following class method for decoding an encoded string:

```java
String decode(String s, String enc)
```

This method decodes an application/x-www-form-urlencoded string using the encoding scheme specified by enc. The supplied encoding is used to determine what characters are represented by any consecutive sequences of the form %xy. UnsupportedEncodingException is thrown when enc’s value isn’t supported.

There are two possible ways in which the decoder could deal with illegally encoded strings. It could either leave illegal characters alone or it could throw IllegalArgumentException. The approach the decoder takes is left to the implementation.
I've created an ED (Encode/Decode) application that demonstrates URLEncoder and URLDecoder in the context of the previous "string ü@foo-bar" and "string+%C3%BC%40foo-bar" example. Listing 12-8 presents the application’s source code.

**Listing 12-8. Encoding and Decoding an Encoded String**

```java
import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;
import java.net.URLEncoder;

public class ED {
    public static void main(String[] args) throws UnsupportedEncodingException {
        String encodedData = URLEncoder.encode("string ü@foo-bar", "UTF-8");
        System.out.println(encodedData);
        System.out.println(URLDecoder.decode(encodedData, "UTF-8");
    }
}
```

When you run this application, it generates the following output:

```
string+%C3%BC%40foo-bar
string ü@foo-bar
```

Note Check out Wikipedia’s “Percent-encoding” topic ([http://en.wikipedia.org/wiki/Percent-encoding](http://en.wikipedia.org/wiki/Percent-encoding)) to learn more about URL encoding (and the more accurate percent-encoding term).

**URI**

The URI class represents URIs (such as URNs and URLs). It doesn’t provide access to a resource when the URI is a URL.

A URI instance stores a character string that conforms to the following syntax at the highest level:

```
[scheme:]scheme-specific-part[#fragment]
```
This syntax reveals that every URI optionally begins with a scheme followed by a colon character, where a scheme can be thought of as an application-level protocol for obtaining an Internet resource. However, this definition is too narrow because it implies that the URI is always a URL. A scheme can have nothing to do with resource location. For example, urn is the scheme for identifying URNs.

A scheme is followed by a scheme-specific-part that provides an instance of the scheme. For example, given the http://tutortutor.ca URI, tutortutor.ca is an instance of the http scheme. Scheme-specific-parts conform to the allowable syntax of their schemes and to the overall syntax structure of a URI (including what characters can be specified literally and what characters must be encoded).

A scheme concludes with an optional #-prefixed fragment, which is a short string of characters that refers to a resource subordinate to another primary resource. The primary resource is identified by a URI; the fragment points to the subordinate resource. For example, http://tutortutor.ca/document.txt#line=5,10 identifies lines 5 through 10 of a text document named document.txt on my web site. (This example is only illustrative; the resource doesn’t actually exist.)

URIs can be categorized as absolute or relative. An absolute URI begins with a scheme followed by a colon character. The earlier http://tutortutor.ca URI is an example of an absolute URI. Other examples include mailto:jeff@tutortutor.ca and news:comp.lang.java.help. Consider an absolute URI as referring to a resource in a manner independent of the context in which that identifier appears. To use a filesystem analogy, an absolute URI is equivalent to a pathname to a file that starts from the root directory.

A relative URI doesn’t begin with a scheme (followed by a colon character). An example is tutorials/tutorials.html. Consider a relative URI as referring to a resource in a manner dependent on the context in which that identifier appears. Using the filesystem analogy, the relative URI is like a pathname to a file that starts from the current directory.

URIs also can be categorized as opaque or hierarchical. An opaque URI is an absolute URI whose scheme-specific part doesn’t begin with a forward slash (/) character. Examples include http://tutortutor.ca and mailto:jeff@tutortutor.ca. Opaque URIs aren’t parsed (beyond identifying their schemes) because scheme-specific parts don’t need to be validated.

A hierarchical URI is either an absolute URI whose scheme-specific part begins with a forward slash character, or is a relative URI. Unlike an opaque URI, a hierarchical URI’s scheme-specific part must be parsed into the various components identified by the following syntax:

```
[/authority] [path] [?query] [#fragment]
```

authority identifies the naming authority for the URI’s namespace. When present, this component begins with a pair of forward slash characters, is either server-based or registry-based, and terminates with the next forward slash character, question mark character, or no more characters—the end of the URI. Registry-based authority components have scheme-specific syntaxes (and aren’t discussed because they’re not commonly used), whereas server-based authority components commonly adopt the following syntax:

```
[userinfo@] host [:port]
```
This syntax specifies that a server-based authority component optionally begins with user information (such as a username) and an “at” (@) character, then continues with the host’s name, and optionally concludes with a colon character and a port. For example, jeff@tutortutor.ca is a server-based authority component, in which jeff denotes the user information and tutortutor.ca denotes the host—there’s no port.

Path identifies the resource’s location according to the authority component (when present) or the scheme (when the authority component is absent). A path divides into a sequence of path segments (portions of the path) in which forward slash characters separate the segments. The path is absolute when the first path segment begins with a forward slash character; otherwise, the path is relative. For example, /a/b/c constitutes a path with three path segments—a, b, and c. Furthermore, the path is absolute because a forward slash character prefixes the first path segment (a).

Query identifies data to be passed to the resource. The resource uses the data to obtain or produce other data that it passes back to the caller. For example, in http://tutortutor.ca/cgi-bin/makepage.cgi?/software/Aquarium, /software/Aquarium represents a query. According to that query, /software/Aquarium is data to be passed to a resource (makepage.cgi), and this data happens to be the absolute path to a directory whose same-named file is merged with boilerplate HTML by a Perl script to generate a resulting web page.

The final component is fragment. Although it appears to be part of a URI, it’s not. When a URI is used in a retrieval action, the primary resource that performs that action uses the fragment to retrieve the subordinate resource. For example, in http://www.example.org/foo.html#bar, foo.html is the primary resource and bar is the subordinate resource (the fragment). Here, #bar is a fragment identifier identifying bar as the subordinate resource.

The previous discussion reveals that a complete URI consists of scheme, authority, path, query, and fragment components; or it consists of scheme, user-info, host, port, path, query, and fragment components. To construct a URI instance in the former case, call the URI(String scheme, String authority, String path, String query, String fragment) constructor. In the latter case, call URI(String scheme, String userInfo, String host, int port, String path, String query, String fragment).

Additional constructors are available for creating URI instances. For example, URI(String uri) creates a URI by parsing uri. Regardless of which constructor you call, it throws java.net.URISyntaxException when the resulting URI string has invalid syntax.

Tip The java.io.File class declares a URI toURI() method that you can call to convert a File object’s abstract pathname to a URI object. The internal URI’s scheme is set to file.

URI declares various getter methods that let you retrieve URI components. For example, String getScheme() lets you retrieve the scheme and String getFragment() returns a URL-decoded fragment. This class also declares boolean isAbsolute() and boolean isOpaque() methods that return true when a URI is absolute and opaque.
Listing 12-9 presents an application that lets you learn about URI components along with absolute and opaque URIs.

**Listing 12-9. Learning About a URI**

```java
import java.net.URI;
import java.net.URISyntaxException;

public class URIComponents
{
    public static void main(String[] args) throws URISyntaxException
    {
        if (args.length != 1)
        {
            System.err.println("usage: java URIComponents uri");
            return;
        }

        URI uri = new URI(args[0]);
        System.out.println("Authority = " + uri.getAuthority());
        System.out.println("Fragment = " + uri.getFragment());
        System.out.println("Host = " + uri.getHost());
        System.out.println("Path = " + uri.getPath());
        System.out.println("Port = " + uri.getPort());
        System.out.println("Query = " + uri.getQuery());
        System.out.println("Scheme = " + uri.getScheme());
        System.out.println("Scheme-specific part = " + uri.getSchemeSpecificPart());
        System.out.println("User Info = " + uri.getUserInfo());
        System.out.println("URI is absolute: " + uri.isAbsolute());
        System.out.println("URI is opaque: " + uri.isOpaque());
    }
}
```

Compile Listing 12-9 (javac URIComponents.java) and execute the following command to run the application:

```
java URIComponents http://tutortutor.ca/cgi-bin/makepage.cgi?/software/Aquarium
```

You’ll observe the following output:

```
Authority = tutortutor.ca
Fragment = null
Host = tutortutor.ca
Path = /cgi-bin/makepage.cgi
Port = -1
Query = /software/Aquarium
Scheme = http
Scheme-specific part = //tutortutor.ca/cgi-bin/makepage.cgi?/software/Aquarium
User Info = null
URI is absolute: true
URI is opaque: false
```
After creating a URI instance, you can perform normalization, resolution, and relativization operations (discussed shortly) on its contained URI. Although you cannot communicate via this instance, you can convert it to a URL instance for communication purposes (assuming that the URI is actually a URL and not a URN or something else) by invoking its URL toURL() method.

This method throws IllegalArgumentException when the URI doesn’t represent an absolute URL, and throws MalformedURLException when a protocol handler for the URL couldn’t be found (i.e., the URL doesn’t start with a supported protocol such as http or file), or when some other error occurred while constructing the URL instance.

**Normalization**

Normalization is the process of removing unnecessary “.” and “..” path segments from a hierarchical URI’s path component. Each “.” segment is removed. A “..” segment is removed only when it’s preceded by a non-“..” segment. Normalization has no effect upon opaque URIs.

URI declares a URI normalize() method for normalizing a URI. This method returns a new URI object that contains the normalized equivalent of its caller’s URI.

Listing 12-10 presents an application that lets you experiment with normalize().

**Listing 12-10. Normalizing URIs**

```java
import java.net.URI;
import java.net.URISyntaxException;

public class Normalize {
    public static void main(String[] args) throws URISyntaxException {
        if (args.length != 1) {
            System.err.println("usage: java Normalize uri");
            return;
        }
        URI uri = new URI(args[0]);
        System.out.println("Normalized URI = " + uri.normalize());
    }
}
```

Compile Listing 12-10 (javac Normalize.java) and run the application as follows:

```
java Normalize a/b/../../../c../d.
```

You should observe the following output, which shows that b isn’t part of a normalized URI:

```
Normalized URI = a/c/d
```
Resolution

Resolution is the process of resolving one URI against another URI, which is known as the base. The resulting URI is constructed from components of both URIs in the manner specified by RFC 2396 (see http://tools.ietf.org/html/rfc2396), taking components from the base URI for those not specified in the original URI. For hierarchical URIs, the path of the original is resolved against the path of the base and then normalized.

For example, the result of resolving original URI docs/guide/collections/designfaq.html#28 against base URI http://docs.oracle.com/javase/1.3/ is result URI http://docs.oracle.com/javase/1.3/docs/guide/collections/designfaq.html#28.

Resolution of both absolute and relative URIs, and of both absolute and relative paths in the case of hierarchical URIs, is supported.

URI declares URI resolve(String str) and URI resolve(URI uri) methods for resolving the original URI argument (passed to str or uri) against the base URI contained in the current URI object (on which this method was called). These methods return either a new URI object containing the original URI or the URI argument when the original URI is already absolute or opaque. Otherwise, they return a new URI object containing the resolved URI. NullPointerException is thrown when str or uri is null. IllegalArgumentException is thrown when str violates RFC 2396 syntax.

Listing 12-11 presents an application that lets you experiment with resolve(String).

Listing 12-11. Resolving URIs

```
import java.net.URI;
import java.net.URISyntaxException;

public class Resolve {
    public static void main(String[] args) throws URISyntaxException {
        if (args.length != 2) {
            System.err.println("usage: java Resolve baseuri uri");
            return;
        }

        URI uri = new URI(args[0]);
        System.out.println("Resolved URI = "+uri.resolve(args[1]));
    }
}
```

Compile Listing 12-11 (javac Resolve.java) and run the application as follows:

```
java Resolve http://docs.oracle.com/javase/1.3/docs/guide/collections/designfaq.html#28
```

You should observe the following output:

```
Resolved URI = http://docs.oracle.com/javase/1.3/docs/guide/collections/designfaq.html#28
```
Relativization

Relativization is the inverse of resolution. For any two normalized URIs, relativization undoes the work performed by resolution and resolution undoes the work performed by relativization.

URI declares a URI relativize(URI uri) method for relativizing its uri argument against the URI in the current URI object (on which this method was called)—relativize() throws NullPointerException when uri is null.

Note For any two normalized URI instances u and v, u.relativize(u.resolve(v)).equals(v) and u.resolve(u.relativize(v)).equals(v) evaluate to true.

relativize() performs relativization of its URI argument’s URI against the URI in the URI object on which this method was called as follows:

- If either this URI or the argument URI is opaque, or if the scheme and authority components of the two URIs aren’t identical, or if the path of this URI isn’t a prefix of the path of the argument URI, the argument URI is returned.
- Otherwise, a new relative hierarchical URI is constructed with query and fragment components taken from the argument URI, and with a path component computed by removing this URI’s path from the beginning of the argument URI’s path.

Listing 12-12 presents an application that lets you experiment with relativize().

Listing 12-12. Relativizing URIs

```java
import java.net.URI;
import java.net.URISyntaxException;

public class Relativize {
    public static void main(String[] args) throws URISyntaxException {
        if (args.length != 2) {
            System.err.println("usage: java Relativize uri1 uri2");
            return;
        }

        URI uri1 = new URI(args[0]);
        URI uri2 = new URI(args[1]);
        System.out.println("Relativized URI = " + uri1.relativize(uri2));
    }
}
```
Compile Listing 12-12 (javac Relativize.java) and run the application as follows: java Relativize http://docs.oracle.com/javase/1.3/ http://docs.oracle.com/javase/1.3/docs/guide/collections/designfaq.html#28. You should observe the following output:

Relativized URI = docs/guide/collections/designfaq.html#28

### Accessing Network Interfaces and Interface Addresses

The `NetworkInterface` class represents a network interface in terms of a name (such as `lo0`) and a list of IP addresses assigned to this interface. Although a network interface is often implemented on a physical NIC, it also can be implemented in software; for example, the `loopback interface` (which is useful for testing a client).

Table 12-1 presents `NetworkInterface`’s methods.

**Table 12-1. NetworkInterface Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object obj)</td>
<td>Compares this <code>NetworkInterface</code> object with obj. The result is true if and only if obj isn't null and represents the same network interface as this object. (Two <code>NetworkInterface</code> objects represent the same network interface when their names and addresses are the same.)</td>
</tr>
<tr>
<td>static NetworkInterface getByInetAddress(InetAddress address)</td>
<td>Returns the <code>NetworkInterface</code> corresponding to the given address or null when no interface has this address. This method throws <code>SocketException</code> when an I/O error occurs and <code>NullPointerException</code> when address is null.</td>
</tr>
<tr>
<td>static NetworkInterface getByName(String interfaceName)</td>
<td>Returns the <code>NetworkInterface</code> with the specified name, or returns null when there's no such network interface. This method throws <code>SocketException</code> on an I/O error and <code>NullPointerException</code> when <code>interfaceName</code> is null.</td>
</tr>
<tr>
<td>String getDisplayName()</td>
<td>Returns this network interface’s <em>display name</em> (a human-readable string describing the network device). On Android, this is the same string as returned by <code>getName()</code>.</td>
</tr>
<tr>
<td>byte[] getHardwareAddress()</td>
<td>Returns an array of bytes containing this network interface’s hardware address, which is often referred to as the <em>media access control (MAC)</em> address. When the interface doesn’t have a MAC address, or when the address cannot be accessed (perhaps the user doesn’t have sufficient privileges), the method returns null. This method throws <code>SocketException</code> when an I/O error occurs.</td>
</tr>
<tr>
<td>Enumeration&lt;InetAddress&gt; getInetAddresses()</td>
<td>Returns an <em>enumeration</em> (the results of an iteration) with all or a subset of the addresses bound to this network interface.</td>
</tr>
<tr>
<td>List&lt;InterfaceAddress&gt; getInterfaceAddresses()</td>
<td>Returns a <code>java.util.List</code> containing this network interface’s <code>InterfaceAddresses</code>.</td>
</tr>
</tbody>
</table>

(continued)
### Method | Description
--- | ---
`int getMTU()` | Returns this network interface’s *maximum transmission unit (MTU).* This method throws SocketException when an I/O error occurs.

`String getName()` | Returns this network interface’s name (such as `eth0` or `lo`).

`static Enumeration<NetworkInterface> getNetworkInterfaces()` | Returns all of the network interfaces on this machine, or returns null when no network interfaces could be found. This method throws SocketException when an I/O error occurs.

`NetworkInterface getParent()` | Returns this network interface’s parent NetworkInterface when this network interface is a subinterface. When this network interface has no parent, or when it’s a physical (nonvirtual) interface, this method returns null. (A physical network interface can be logically divided into multiple virtual subinterfaces, which are commonly used in routing and switching. These subinterfaces can be organized into a hierarchy where the physical network interface serves as the root.)

`Enumeration<NetworkInterface> getSubInterfaces()` | Returns an enumeration containing the virtual subinterfaces that are attached to this network interface. For example, `eth0:1` is a subinterface of `eth0`.

`int hashCode()` | This method is overridden because `equals()` is overridden.

`boolean isLoopback()` | Returns true when this network interface reflects outgoing data back to itself as incoming data. This method throws SocketException when an I/O error occurs.

`boolean isPointToPoint()` | Returns true when this network interface is point-to-point (such as a PPP connection through a modem). This method throws SocketException when an I/O error occurs.

`boolean isUp()` | Returns true when this network interface is *up* (routing entries have been established) and *running* (platform resources have been allocated). This method throws SocketException when an I/O error occurs.

`boolean isVirtual()` | Returns true when this network interface is a virtual subinterface. On some platforms, virtual subinterfaces are network interfaces created as children of a physical network interface and given different settings (such as address or MTU). Usually, the name of the interface will be the name of the parent followed by a colon (:) and a number identifying the child because there can be several virtual subinterfaces attached to a single physical network interface.

`boolean supportsMulticast()` | Returns true when this network interface supports *multicasting.* This method throws SocketException when an I/O error occurs.

`String toString()` | Returns a string representation of this network interface.
You can use these methods to gather useful information about your platform’s network interfaces. For example, Listing 12-13 presents an application that iterates over all network interfaces, invoking the methods listed in Table 12-1 that obtain the network interface’s name and display name, determine if the network interface is a loopback interface, determine if the network interface is up and running, obtain the MTU, determine if the network interface supports multicasting, and enumerate all of the network interface’s virtual subinterfaces.

Listing 12-13. Enumerating All Network Interfaces

```java
import java.net.NetworkInterface;
import java.net.SocketException;

import java.util.Collections;
import java.util.Enumeration;

public class NetInfo
{
    public static void main(String[] args) throws SocketException
    {
        Enumeration<NetworkInterface> eni;
        eni = NetworkInterface.getNetworkInterfaces();
        for (NetworkInterface ni: Collections.list(eni))
        {
            System.out.println("Name = " + ni.getName());
            System.out.println("Display Name = " + ni.getDisplayName());
            System.out.println("Loopback = " + ni.isLoopback());
            System.out.println("Up and running = " + ni.isUp());
            System.out.println("MTU = " + ni.getMTU());
            System.out.println("Supports multicast = " + ni.supportsMulticast());
            System.out.println("Sub-interfaces");
            Enumeration<NetworkInterface> eni2;
            eni2 = ni.getSubInterfaces();
            for (NetworkInterface ni2: Collections.list(eni2))
            {
                System.out.println("   " + ni2);
            }
            System.out.println();
        }
    }
}
```

Tip The java.util.Collections class's ArrayList<T> list(Enumeration<T> enumeration) method is useful for converting a legacy enumeration to a modern array list.

Compile Listing 12-13 (javac NetInfo.java) and execute this application (java NetInfo). When I run NetInfo on my Windows 7 platform, I observe information that begins with the following output:

Name = lo
Display Name = Software Loopback Interface 1
Loopback = true
CHAPTER 12: Accessing Networks

Up and running = true  
MTU = -1  
Supports multicast = true  
Sub-interfaces

Name = net0  
Display Name = WAN Miniport (SSTP)  
Loopback = false  
Up and running = false  
MTU = -1  
Supports multicast = true  
Sub-interfaces

The complete output reveals a different MTU size for a few network interfaces. Each size represents the maximum length of a message that can fit into an IP datagram without needing to fragment the message into multiple IP datagrams. This fragmentation has performance implications, especially in the context of networked games. For this reason alone, the getMTU() method is a valuable member of NetworkInterface.

The getInterfaceAddresses() method returns a list of InterfaceAddress objects, with each object containing a network interface’s IP address along with broadcast address and subnet mask (IPv4) or network prefix length (IPv6).

Table 12-2 presents InterfaceAddress’s methods.

**Table 12-2. InterfaceAddress Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object obj)</td>
<td>Compares this InterfaceAddress object with obj. Returns true when obj is also an InterfaceAddress, and when both objects contain the same InetAddress, the same subnet masks/network prefix lengths (depending on IPv4 or IPv6), and the same broadcast addresses.</td>
</tr>
<tr>
<td>InetAddress getAddress()</td>
<td>Returns this InterfaceAddress’s IP address, as an InetAddress object.</td>
</tr>
<tr>
<td>InetAddress getBroadcast()</td>
<td>Returns this InterfaceAddress’s broadcast address (IPv4) or null (IPv6); IPv6 doesn’t support broadcast addresses.</td>
</tr>
<tr>
<td>short getNetworkPrefixLength()</td>
<td>Returns this InterfaceAddress’s network prefix length (IPv6) or subnet mask (IPv4). Oracle’s Java documentation shows 128 (::1/128) and 10 (fe80::203:baff:fe27:1243/10) as typical IPv6 values. Typical IPv4 values are 8 (255.0.0.0), 16 (255.255.0.0), and 24 (255.255.255.0).</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns this InterfaceAddress’s hash code. The hash code is a combination of the InetAddress’s hash code, the broadcast address (when present) hash code, and the network prefix length.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string representation of this InterfaceAddress. This representation has the form InetAddress / network prefix length [broadcast address].</td>
</tr>
</tbody>
</table>
Listing 12-14, which extends Listing 12-13 (with a few lines removed), enumerates all network interfaces, outputting their display names, and enumerates each network interface’s interface addresses, outputting interface address information.

Listing 12-14. Enumerating All Network Interfaces and Interface Addresses

```java
import java.net.InterfaceAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;

public class NetInfo
{
    public static void main(String[] args) throws SocketException
    {
        Enumeration<NetworkInterface> eni;
        eni = NetworkInterface.getNetworkInterfaces();
        for (NetworkInterface ni: Collections.list(eni))
        {
            System.out.println("Name = " + ni.getName());
            List<InterfaceAddress> ias = ni.getInterfaceAddresses();
            Iterator<InterfaceAddress> iter = ias.iterator();
            while (iter.hasNext())
            {
                System.out.println(iter.next());
            }
            System.out.println();
        }
    }
}
```

Compile Listing 12-14 (javac NetInfo.java) and execute this application (java NetInfo). When I run NetInfo on my Windows 7 platform, I observe the following information:

Name = lo
/127.0.0.1/8 [/127.255.255.255]
/0:0:0:0:0:0:1/128 [null]

Name = net0
Name = net1
Name = net2
Name = ppp0
Name = eth0
Name = eth1
Name = eth2
Name = ppp1
Name = net3
Name = eth3
/192.xxx.xxx.xxx/xx [/192.xxx.xxx.xxx]
/fe80:0:0:0:xxxx:xxxx:xxxx:xxxx%xx [null]
Name = net4
/fe80:0:0:0:xxxx:xxxx%xx/xxx [null]
Name = net5
/fe80:0:0:0:xxxx:xxxx:xxxx:xxxx%xx/xx [null]
Name = eth4
Name = eth5
Name = eth6
Name = eth7
Name = eth8

Managing Cookies

Server applications commonly use HTTP cookies (state objects)—cookies for short—to persist small amounts of information on clients. For example, the identifiers of currently selected items in a shopping cart can be stored as cookies. It’s preferable to store cookies on the client rather than on the server because of the potential for millions of cookies (depending on a web site’s popularity). In that case, not only would a server require a massive amount of storage just for cookies, but also searching for and maintaining cookies would be time consuming.

Note Check out Wikipedia’s “HTTP cookie” entry (http://en.wikipedia.org/wiki/HTTP_cookie) for a quick refresher on cookies.

A server application sends a cookie to a client as part of an HTTP response. A client (such as a web browser) sends a cookie to the server as part of an HTTP request. Before Java 5, applications worked with the URLConnection class (and its HttpURLConnection subclass) to get an HTTP response's cookies and to set an HTTP request's cookies. The String getHeaderFieldKey(int n) and String getHeaderField(int n) methods were used to access a response’s Set-Cookie headers, and the void setRequestProperty(String key, String value) method was used to create a request’s Cookie header.
Java 5 introduced the abstract CookieHandler class as a callback mechanism that connects HTTP state management to an HTTP protocol handler (think concrete HttpURLConnection subclass). An application installs a concrete CookieHandler subclass as the system-wide cookie handler via the CookieHandler class's void setDefault(CookieHandler cHandler) class method. A companion CookieHandler getDefault() class method returns this cookie handler, which is null when a system-wide cookie handler hasn't been installed.

An HTTP protocol handler accesses response and request headers. This handler invokes the system-wide cookie handler's void put(URI uri, Map<String, List<String>> responseHeaders) method to store response cookies in a cookie cache and invokes the Map<String, List<String>> get(URI uri, Map<String, List<String>> requestHeaders) method to fetch request cookies from this cache. Unlike Java 5, Java 6 introduced a concrete implementation of CookieHandler so that HTTP protocol handlers and applications can work with cookies.

The concrete CookieManager class extends CookieHandler to manage cookies. This class does not interact with the web browser's cookies (stored on the client computer). Instead, it represents a separate and distinct cookie manager.

A CookieManager object is initialized as follows:

- With a cookie store for storing cookies. The cookie store is based on the CookieStore interface.
- With a cookie policy for determining which cookies to accept for storage. The cookie policy is based on the CookiePolicy interface.

Create a cookie manager by calling either the CookieManager() constructor or the CookieManager(CookieStore store, CookiePolicy policy) constructor. The CookieManager() constructor invokes the latter constructor with null arguments, using the default in-memory cookie store and the default accept-cookies-from-the-original-server-only cookie policy. Unless you plan to create your own CookieStore and CookiePolicy implementations, you'll most likely work with the default constructor. The following example creates and establishes a new CookieManager object as the system-wide cookie handler:

```java
CookieHandler.setDefault(new CookieManager());
```

Along with the aforementioned constructors, CookieManager declares the following methods:

- Map<String, List<String>> get(URI uri, Map<String, List<String>> requestHeaders) returns an immutable map of Cookie and Cookie2 request headers for cookies obtained from the cookie store whose path matches uri's path. Although requestHeaders isn't used by the default implementation of this method, it can be used by subclasses. IOException is thrown when an I/O error occurs.

- CookieStore getCookieStore() returns the cookie manager's cookie store.
void put(URI uri, Map<String, List<String>> responseHeaders) stores all applicable cookies whose Set-Cookie and Set-Cookie2 response headers were retrieved from the specified uri value and placed (with all other response headers) in the immutable responseHeaders map in the cookie store. IOException is thrown when an I/O error occurs.

void setCookiePolicy(CookiePolicy cookiePolicy) sets the cookie manager's cookie policy to one of CookiePolicy.ACCEPT_ALL (accept all cookies), CookiePolicy.ACCEPT_NONE (accept no cookies), or CookiePolicy.ACCEPT_ORIGINAL_SERVER (accept cookies from original server only—this is the default). Passing null to this method has no effect on the current policy.

In contrast to the get() and put() methods, which are called by HTTP protocol handlers, an application works with the getCookieStore() and setCookiePolicy() methods. Consider Listing 12-15.

Listing 12-15. Listing All Cookies for a Specific Domain

import java.io.IOException;
import java.net.CookieHandler;
import java.net.CookieManager;
import java.net.CookiePolicy;
import java.net.HttpCookie;
import java.net.URL;
import java.util.List;

public class ListAllCookies
{
    public static void main(String[] args) throws IOException
    {
        if (args.length != 1)
        {
            System.err.println("usage: java ListAllCookies url");
            return;
        }
        CookieManager cm = new CookieManager();
        cm.setCookiePolicy(CookiePolicy.ACCEPT_ALL);
        CookieHandler.setDefault(cm);
        new URL(args[0]).openConnection().getContent();
        List<HttpCookie> cookies = cm.getCookieStore().getCookies();
        for (HttpCookie cookie : cookies)
        {
            System.out.println("Name = " + cookie.getName());
            System.out.println("Value = " + cookie.getValue());
            System.out.println(" Lifetime (seconds) = " + cookie.getMaxAge());
            System.out.println(" Path = " + cookie.getPath());
        }
    }
}
Listing 12-15 describes a command-line application that obtains and lists all cookies from its single domain name argument. After creating a cookie manager and invoking `setCookiePolicy()` to set the cookie manager's policy to accept all cookies, `ListAllCookies` installs the cookie manager as the system-wide cookie handler. It next connects to the domain identified by the command-line argument and reads the content (via URL's `Object getContent()` method).

The cookie store is obtained via `getCookieStore()` and used to retrieve all nonexpired cookies via its `List<HttpCookie> getCookies()` method. For each of these HttpCookies, `String getName()`, `String getValue()`, and other `HttpCookie` methods are invoked to return cookie-specific information.

Compile Listing 12-15 (`javac ListAllCookies.java`). The following output resulted from invoking `java ListAllCookies http://java.dzone.com`:

```
Name = SESS374e8db54ec3033c25a586b1d093b1d1
Value = irhqtiemls4cp0vf5pe1p0oeo7
Lifetime (seconds) = 2000000
Path = /
```

**Note**  For more information about cookie management, including examples that show you how to create your own `CookiePolicy` and `CookieStore` implementations, check out *The Java Tutorial*'s “Working With Cookies” lesson ([http://docs.oracle.com/javase/tutorial/networking/cookies/index.html](http://docs.oracle.com/javase/tutorial/networking/cookies/index.html)).

### EXERCISES

The following exercises are designed to test your understanding of Chapter 12's content.

1. Define network.
2. What is an intranet and what is an internet?
3. What do intranets and internets often use to communicate between nodes?
4. Define host.
5. What is a socket?
6. How is a socket identified?
7. Define IP address.
8. What is a packet?
9. A socket address is comprised of what elements?
10. Identify the `InetAddress` subclasses that are used to represent IPv4 and IPv6 addresses.
11. What is the loopback interface?
12. True or false: In network byte order, the least significant byte comes first.
13. How is the local host represented?
15. How are socket options described?
16. True or false: You set a socket option by calling the `void setOption(int optID, Object value)` method.
17. Why are sockets based on the `Socket` class commonly referred to as stream sockets?
18. What does binding accomplish in the context of a `Socket` instance?
19. Define proxy. How does Java represent proxy settings?
20. True or false: The `ServerSocket()` constructor creates a bound sever socket.
21. What is the difference between the `DatagramSocket` and `MulticastSocket` classes?
22. What is a datagram packet?
23. What is the difference between unicasting and multicasting?
24. What is a URL?
25. What is a URN?
26. True or false: URLs and URNs are also URIs.
27. What does the `URL(String s)` constructor do when you pass `null` to `s`?
28. What is the equivalent of `openStream()`?
29. True or false: You need to invoke `URLConnection`'s `void setDoInput(boolean doInput)` method with `true` as the argument before you can input content from a web resource.
30. What does `URLEncoder` do when it encounters a space character?
31. What is the purpose of the `URI` class?
32. Define normalization.
33. True or false: Resolution and relativization are inverse operations of each other.
34. What does the `NetworkInterface` class accomplish?
35. What is a MAC address?
36. What does MTU stand for and what is its purpose?
37. True or false: `NetworkInterface`'s `getName()` method returns a human-readable name.
38. What does `InterfaceAddress`'s `getNetworkPrefixLength()` method return under IPv4?
39. Define HTTP Cookie.
40. Why is it preferable to store cookies on the client rather than on the server?
41. Identify the four `java.net` types that are used to work with cookies.
42. Modify Listing 12-1’s EchoClient source code to explicitly close the socket.

43. Modify Listing 12-2’s EchoServer source code to exit the while loop and explicitly close the server socket when a file named kill appears in the directory from which the server was started. After this file appears, the server will probably not die immediately because it’s most likely waiting (via the accept() call) for an incoming client connection. However, it should die after servicing the next incoming connection.

Summary

A network is a group of interconnected nodes that can be shared among the network’s users. An intranet is a network located within an organization and an internet is a network connecting organizations to each other. The Internet is the global network of networks.

The java.net package provides types that support TCP/IP between processes running on the same or different hosts. Two processes communicate by way of sockets, which are endpoints in a communications link between these processes. Each endpoint is identified by an IP address that identifies a host and by a port number that identifies the process running on that host.

One process writes a message to its socket, the network management software portion of the underlying operating system breaks the message into a sequence of packets that it forwards to the other process’s socket, and the other process recombines received packets into the original message for its own processing.

The network management software uses TCP to create an ongoing conversation between two hosts in which messages are sent back and forth. Before this conversation occurs, a connection is established between these hosts. After the connection has been established, TCP enters a pattern where it sends message packets and waits for a reply that they arrived correctly (or for a timeout to expire when the reply doesn’t arrive because of some network problem). This pattern repeats and guarantees a reliable connection.

Because it can take time to establish a connection, and it also takes time to send packets (as it is necessary to receive reply acknowledgments, and also because of timeouts), TCP is slow. On the other hand, UDP, which doesn’t require connections and packet acknowledgement, is much faster. The downside is that UDP isn’t as reliable (there’s no guarantee of packet delivery, ordering, or protection against duplicate packets, although UDP uses checksums to verify that data is correct) because there’s no acknowledgment. Furthermore, UDP is limited to single-packet conversations.

An instance of a Socket-suffixed class is associated with a socket address comprised of an IP address and a port number. These classes often rely on the InetAddress class to represent the IPv4 or IPv6 address portion of the socket address, and represent the port number separately.

An instance of a Socket-suffixed class shares the concept of socket options, which are parameters for configuring socket behavior. Socket options are described by constants that are declared in the SocketOptions interface.

The Socket and ServerSocket classes support TCP-based communications between client processes and server processes. Socket supports the creation of client-side sockets, whereas ServerSocket supports the creation of server-side sockets.
The DatagramSocket and MulticastSocket classes let you perform UDP-based communications between a pair of hosts (DatagramSocket) or between as many hosts as necessary (MulticastSocket). With either class, you communicate one-way messages via datagram packets.

Two processes communicating via sockets demonstrate low-level network access. Java also supports high-level access via URLs that identify resources and specify where they are located on TCP/IP-based networks.

URLs are represented by the URL class, which provides access to the resources to which they refer. URLConnection gives you additional control over client/server communication. For example, you can use this class to output content to various resources that accept content. In contrast, URL only lets you input content via openStream().

HTML lets you introduce forms into web pages that solicit information from page visitors. The java.net package provides URLEncoder and URLDecoder classes to assist you with the tasks of encoding and decoding form content.

URLs are a form of URI, which is a character string that identifies a resource without providing access to the resource, or identifies a name. URIs are represented by the URI class, which provides methods for extracting parts of a URI (such as the scheme), and for performing normalization, resolution, and relativization operations.

The NetworkInterface class represents a network interface in terms of a name (such as le0) and a list of IP addresses assigned to this interface. NetworkInterface's getInterfaceAddresses() method returns a list of InterfaceAddress objects, with each object containing a network interface's IP address along with broadcast address and subnet mask (IPv4) or network prefix length (IPv6).

Server applications commonly use HTTP cookies (state objects)—cookies, for short—to persist small amounts of information on clients. Java provides the CookieHandler and CookieManager classes and the CookiePolicy and CookieStore interfaces for working with cookies.

This chapter focused on I/O in a network context. New I/O lets you perform file-based and network-based I/O in a more performant manner. Chapter 13 introduces you to Java's New I/O APIs.
Chapter 13

Migrating to New I/O

Chapters 11 and 12 introduced you to Java’s classic I/O APIs. Chapter 11 presented classic I/O in terms of java.io’s File, RandomAccessFile, stream, and writer/reader types. Chapter 12 presented classic I/O in terms of java.net’s socket and URL types.

Modern operating systems offer powerful I/O features that are not supported by Java’s classic I/O APIs. Features include memory-mapped file I/O (the ability to map part of a process (executing application)’s virtual memory (see http://en.wikipedia.org/wiki/Virtual_memory) to some portion of a file so that writes to or reads from that portion of the process’s memory space actually write/read the associated portion of the file), readiness selection (a step above nonblocking I/O that offloads to the operating system the work involved in checking for I/O stream readiness to perform write and read operations), and file locking (the ability for one process to prevent other processes from accessing a file or to limit the access in some way).

Java 1.4 introduced a more powerful I/O architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture largely consists of buffers, channels, selectors, regular expressions, and charsets, and it is commonly known as New I/O (NIO).

Note  Regular expressions are included as part of NIO (see JSR 51 at http://jcp.org/en/jsr/detail?id=51) because NIO is all about performance, and regular expressions are useful for scanning text (read from an I/O source) in a highly performant manner.

In this chapter, I introduce you to NIO in terms of buffers, channels, selectors, regular expressions, and charsets. I also discuss the simple printf-style formatting facility proposed in JSR 51 but not implemented until Java 5.
Working with Buffers

NIO is based on buffers. A buffer is an object that stores a fixed amount of data to be sent to or received from an I/O service (a means for performing input/output). It sits between an application and a channel that writes the buffered data to the service or reads the data from the service and deposits it into the buffer.

Buffers possess four properties:

- **Capacity**: The total number of data items that can be stored in the buffer. The capacity is specified when the buffer is created and cannot be changed later.
- **Limit**: The number of “live” data items in the buffer. No items starting from the zero-based limit should be written or read.
- **Position**: The zero-based index of the next data item that can be read or the location where the data item can be written.
- **Mark**: A zero-based position that can be recalled. The mark is initially undefined.

These four properties are related as follows:

\[ 0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity} \]

Figure 13-1 reveals a newly created and byte-oriented buffer with a capacity of 7.

Figure 13-1’s buffer can store a maximum of seven elements. The mark is initially undefined, the position is initially set to 0, and the limit is initially set to the capacity (7), which specifies the maximum number of bytes that can be stored in the buffer. You can only access positions 0 through 6. Position 7 lies beyond the buffer.

Buffer and Its Children

Buffers are implemented by classes that are derived from the abstract `java.nio.Buffer` class. Table 13-1 describes Buffer’s methods.
### Table 13-1. Buffer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object array()</code></td>
<td>Returns the array that backs this buffer. This method is intended to allow array-backed buffers to be passed to native code more efficiently. Concrete subclasses override this method and provide more strongly typed return values via covariant return types (discussed in Chapter 4). This method throws <code>java.nio.ReadOnlyBufferException</code> when this buffer is backed by an array but is read only and throws <code>java.lang.UnsupportedOperationException</code> when this buffer isn’t backed by an accessible array.</td>
</tr>
<tr>
<td><code>int arrayOffset()</code></td>
<td>Returns the offset of the first buffer element within this buffer’s backing array. When this buffer is backed by an array, buffer position p corresponds to array index p + arrayOffset(). Invoke <code>hasArray()</code> before invoking this method to ensure that this buffer has an accessible backing array. This method throws <code>ReadOnlyBufferException</code> when this buffer is backed by an array but is read only and throws <code>UnsupportedOperationException</code> when this buffer isn’t backed by an accessible array.</td>
</tr>
<tr>
<td><code>int capacity()</code></td>
<td>Returns this buffer’s capacity.</td>
</tr>
<tr>
<td><code>Buffer clear()</code></td>
<td>Clears this buffer. The position is set to 0, the limit is set to the capacity, and the mark is discarded. This method doesn’t erase the data in the buffer but is named as if it did because it will most often be used in situations in which that might as well be the case.</td>
</tr>
<tr>
<td><code>Buffer flip()</code></td>
<td>Flips this buffer. The limit is set to the current position and then the position is set to 0. When the mark is defined, it’s discarded.</td>
</tr>
<tr>
<td><code>boolean hasArray()</code></td>
<td>Returns true when this buffer is backed by an array and isn’t read-only; otherwise, returns false. When this method returns true, <code>array()</code> and <code>arrayOffset()</code> may be invoked safely.</td>
</tr>
<tr>
<td><code>boolean hasRemaining()</code></td>
<td>Returns true when at least one element remains in this buffer (that is, between the current position and the limit); otherwise, returns false.</td>
</tr>
<tr>
<td><code>boolean isDirect()</code></td>
<td>Returns true when this buffer is a direct byte buffer (discussed later in this section); otherwise, returns false.</td>
</tr>
<tr>
<td><code>boolean isReadOnly()</code></td>
<td>Returns true when this buffer is read-only; otherwise, returns false.</td>
</tr>
<tr>
<td><code>int limit()</code></td>
<td>Returns this buffer’s limit.</td>
</tr>
<tr>
<td><code>Buffer limit(int newLimit)</code></td>
<td>Sets this buffer’s limit to <code>newLimit</code>. When the position is larger than <code>newLimit</code>, the position is set to <code>newLimit</code>. When the mark is defined and is larger than <code>newLimit</code>, the mark is discarded. This method throws <code>java.lang.IllegalArgumentException</code> when <code>newLimit</code> is negative or larger than this buffer’s capacity; otherwise, it returns this buffer.</td>
</tr>
<tr>
<td><code>Buffer mark()</code></td>
<td>Sets this buffer’s mark to its position and returns this buffer.</td>
</tr>
<tr>
<td><code>int position()</code></td>
<td>Returns this buffer’s position.</td>
</tr>
</tbody>
</table>
Table 13-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer position(int newPosition)</td>
<td>Sets this buffer’s position to newPosition. When the mark is defined and is larger than newPosition, the mark is discarded. This method throws IllegalArgumentException when newPosition is negative or larger than this buffer’s current limit; otherwise, it returns this buffer.</td>
</tr>
<tr>
<td>int remaining()</td>
<td>Returns the number of elements between the current position and the limit.</td>
</tr>
<tr>
<td>Buffer reset()</td>
<td>Resets this buffer’s position to the previously marked position. Invoking this method neither changes nor discards the mark’s value. This method throws java.nio.InvalidMarkException when the mark hasn’t been set; otherwise, it returns this buffer.</td>
</tr>
<tr>
<td>Buffer rewind()</td>
<td>Rewinds and then returns this buffer. The position is set to 0 and the mark is discarded.</td>
</tr>
</tbody>
</table>

Table 13-1 shows that many of Buffer’s methods return Buffer references so that you can chain instance method calls together. (See Chapter 3 for a discussion on instance method call chaining.) For example, instead of specifying the following three lines,

```java
buf.mark();
buf.position(2);
buf.reset();
```

you can more conveniently specify the following line:

```java
buf.mark().position(2).reset();
```

Table 13-1 also shows that all buffers can be read but not all buffers can be written—for example, a buffer backed by a memory-mapped file that’s read-only. You must not write to a read-only buffer; otherwise, ReadOnlyBufferException is thrown. Call isReadOnly() when you’re unsure that a buffer is writable before attempting to write to that buffer.

**Caution** Buffers are not thread-safe. You must employ synchronization when you want to access a buffer from multiple threads.

The java.nio package includes several abstract classes that extend Buffer, one for each primitive type except for Boolean: ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer. Furthermore, this package includes MappedByteBuffer as an abstract ByteBuffer subclass.
Note Operating systems perform byte-oriented I/O, and you use ByteBuffer to create byte-oriented buffers that store the bytes to write to a destination or that are read from a source. The other primitive-type buffer classes let you create multibyte view buffers (discussed later) so that you can conceptually perform I/O in terms of characters, double precision floating-point values, 32-bit integers, and so on. However, the I/O operation is really being carried out as a flow of bytes.

Listing 13-1 demonstrates the Buffer class in terms of ByteBuffer, capacity, limit, position, and remaining elements.

Listing 13-1. Demonstrating a Byte-Oriented Buffer

```java
import java.nio.Buffer;
import java.nio.ByteBuffer;

public class BufferDemo
{
    public static void main(String[] args)
    {
        Buffer buffer = ByteBuffer.allocate(7);
        System.out.println("Capacity: " + buffer.capacity());
        System.out.println("Limit: " + buffer.limit());
        System.out.println("Position: " + buffer.position());
        System.out.println("Remaining: " + buffer.remaining());
        System.out.println("Changing buffer limit to 5");
        buffer.limit(5);
        System.out.println("Limit: " + buffer.limit());
        System.out.println("Position: " + buffer.position());
        System.out.println("Remaining: " + buffer.remaining());
        System.out.println("Changing buffer position to 3");
        buffer.position(3);
        System.out.println("Position: " + buffer.position());
        System.out.println("Remaining: " + buffer.remaining());
        System.out.println(buffer);
    }
}
```

Listing 13-1’s main() method first needs to obtain a buffer. It cannot instantiate the Buffer class because that class is abstract. Instead, it uses the ByteBuffer class and its allocate() class method to allocate the 7-byte buffer shown in Figure 13-1. main() then calls assorted Buffer methods to demonstrate capacity, limit, position, and remaining elements.

Compile Listing 13-1 as follows:

```
javac BufferDemo.java
```

Run this application as follows:

```
java BufferDemo
```
You should observe the following output:

| Capacity: 7  |
| Limit: 7    |
| Position: 0 |
| Remaining: 7|
| Changing buffer limit to 5
| Limit: 5    |
| Position: 0 |
| Remaining: 5|
| Changing buffer position to 3
| Position: 3 |
| Remaining: 2|
| java.nio.HeapByteBuffer[pos=3 lim=5 cap=7] |

The final output line reveals that the ByteBuffer instance assigned to buffer is actually an instance of the package-private java.nio.HeapByteBuffer class.

**Buffers in Depth**

The previous discussion of the Buffer class has given you some insight into NIO buffers. However, there's much more to explore. This section takes you deeper into buffers by exploring buffer creation, buffer writing and reading, buffer flipping, buffer marking, Buffer subclass operations, byte ordering, and direct buffers.

**Note** Although the primitive-type buffer classes have similar capabilities, ByteBuffer is the largest and most versatile. After all, bytes are the basic unit used by operating systems to transfer data items. I'll therefore use ByteBuffer to demonstrate most buffer operations. I'll also use CharBuffer to add variety.

**Buffer Creation**

ByteBuffer and the other primitive-type buffer classes declare various class methods for creating a buffer of that type. For example, ByteBuffer declares the following class methods for creating ByteBuffer instances:

- ByteBuffer allocate(int capacity): Allocates a new byte buffer with the specified capacity value. Its position is 0, its limit is its capacity, its mark is undefined, and each element is initialized to 0. It has a backing array, and its array offset is 0. This method throws IllegalArgumentException when capacity is negative.

- ByteBuffer allocateDirect(int capacity): Allocates a new direct byte buffer with the specified capacity value. Its position is 0, its limit is its capacity, its mark is undefined, and each element is initialized to 0. Whether or not it has a backing array is unspecified. This method throws IllegalArgumentException when capacity is negative.
ByteBuffer wrap(byte[] array): Wraps a byte array into a buffer. The new buffer is backed by array; that is, modifications to the buffer will cause the array to be modified and vice versa. The new buffer's capacity and limit are set to array.length, its position is set to 0, and its mark is undefined. Its array offset is 0.

ByteBuffer wrap(byte[] array, int offset, int length): Wraps a byte array into a buffer. The new buffer is backed by array. The new buffer's capacity is set to array.length, its position is set to offset, its limit is set to offset + length, and its mark is undefined. Its array offset is 0. This method throws java.lang.IndexOutOfBoundsException when offset is negative or greater than array.length or when length is negative or greater than array.length - offset.

These methods show two ways to create a byte buffer: create the ByteBuffer object and allocate an internal array that stores capacity bytes or create the ByteBuffer object and use the specified array to store these bytes. Consider these examples:

```
ByteBuffer buffer = ByteBuffer.allocate(10);
byte[] bytes = new byte[200];
ByteBuffer buffer2 = ByteBuffer.wrap(bytes);
```

The first line creates a byte buffer with an internal byte array that stores a maximum of 10 bytes, and the second and third lines create a byte array and a byte buffer that uses this array to store a maximum of 200 bytes.

Now consider the following example, which extends the previous example:

```
buffer = ByteBuffer.wrap(bytes, 10, 50);
```

This example creates a byte buffer with a position of 10, a limit of 50, and a capacity of bytes.length (which happens to be 200). Although it appears that the buffer can only access a subrange of this array, it actually has access to the entire array: values 10 and 50 are only the starting values for the position and limit.

ByteBuffers (and other primitive type buffers) created via allocate() or wrap() are nondirect byte buffers; you'll learn about direct byte buffers later. Nondirect byte buffers have backing arrays, and you can access these backing arrays via the array() method (which happens to be declared as byte[] array() in the ByteArray class) as long as hasArray() returns true. (When hasArray() returns true, you'll need to call arrayOffset() to obtain the location of the first data item in the array.)

Listing 13-2 demonstrates buffer allocation and wrapping.

Listing 13-2. Creating Byte-Oriented Buffers via Allocation and Wrapping

```java
import java.nio.ByteBuffer;

public class BufferDemo
{
    public static void main(String[] args)
    {
        ByteBuffer buffer1 = ByteBuffer.allocate(10);
        if (buffer1.hasArray())
```
{  
    System.out.println("buffer1 array: " + buffer1.array());  
    System.out.println("Buffer1 array offset: " + buffer1.arrayOffset());  
    System.out.println("Capacity: " + buffer1.capacity());  
    System.out.println("Limit: " + buffer1.limit());  
    System.out.println("Position: " + buffer1.position());  
    System.out.println("Remaining: " + buffer1.remaining());  
    System.out.println();  
}

byte[] bytes = new byte[200];  
ByteBuffer buffer2 = ByteBuffer.wrap(bytes);  
buffer2 = ByteBuffer.wrap(bytes, 10, 50);  
if (buffer2.hasArray())  
{
    System.out.println("buffer2 array: " + buffer2.array());  
    System.out.println("Buffer2 array offset: " + buffer2.arrayOffset());  
    System.out.println("Capacity: " + buffer2.capacity());  
    System.out.println("Limit: " + buffer2.limit());  
    System.out.println("Position: " + buffer2.position());  
    System.out.println("Remaining: " + buffer2.remaining());  
}

}

Compile Listing 13-2 (javac BufferDemo.java), and run this application (java BufferDemo). You should observe the following output:

buffer1 array: [B@15e565bd  
Buffer1 array offset: 0  
Capacity: 10  
Limit: 10  
Position: 0  
Remaining: 10

buffer2 array: [B@77a6686  
Buffer2 array offset: 0  
Capacity: 200  
Limit: 60  
Position: 10  
Remaining: 50

In addition to managing data elements stored in external arrays (via the wrap() methods), buffers can also manage data stored in other buffers. When you create a buffer that manages another buffer’s data, the created buffer is known as a view buffer. Changes made in either buffer are reflected in the other.
View buffers are created by calling a Buffer subclass’s duplicate() method. The resulting view buffer is equivalent to the original buffer; both buffers share the same data items and have equivalent capacities. However, each buffer has its own position, limit, and mark. When the buffer being duplicated is read-only or direct, the view buffer is also read-only or direct.

Consider the following example:

```java
ByteBuffer buffer = ByteBuffer.allocate(10);
ByteBuffer bufferView = buffer.duplicate();
```

The ByteBuffer instance identified by bufferView shares the same internal array of 10 elements as buffer. At the moment, these buffers have the same position, limit, and (undefined) mark. However, these properties in one buffer can be changed independently of the properties in the other buffer.

View buffers are also created by calling one of ByteBuffer’s asXBuffer() methods. For example, LongBuffer asLongBuffer() returns a view buffer that conceptualizes the byte buffer as a buffer of long integers.

**Note**  
Read-only view buffers can be created by calling a method such as ByteBuffer asReadOnlyBuffer(). Any attempt to change a read-only view buffer’s content results in ReadOnlyBufferException. However, the original buffer content (provided that it isn’t read-only) can be changed, and the read-only view buffer will reflect these changes.

**Buffer Writing and Reading**

ByteBuffer and the other primitive-type buffer classes declare several overloaded put() and get() methods for writing data items to and reading data items from a buffer. These methods are absolute when they require an index argument or relative when they don’t require an index.

For example, ByteBuffer declares the absolute ByteBuffer put(int index, byte b) method to store byte b in the buffer at the index value and the absolute byte get(int index) method to fetch the byte located at position index. This class also declares the relative ByteBuffer put(byte b) method to store byte b in the buffer at the current position and then increment the current position, and the relative byte get() method to fetch the byte located at the buffer’s current position and increment the current position.

The absolute put() and get() methods throw IndexOutOfBoundsException when index is negative or greater than or equal to the buffer’s limit. The relative put() method throws java.nio.BufferOverflowException when the current position is greater than or equal to the limit, and the relative get() method throws java.nio.BufferUnderflowException when the current position is greater than or equal to the limit. Furthermore, the absolute and relative put() methods throw ReadOnlyBufferException when the buffer is read-only.
Listing 13-3 demonstrates the relative put() method and the absolute get() method.

Listing 13-3. Writing Bytes to and Reading Them from a Buffer

import java.nio.ByteBuffer;

public class BufferDemo
{
    public static void main(String[] args)
    {
        ByteBuffer buffer = ByteBuffer.allocate(7);
        System.out.println("Capacity = " + buffer.capacity());
        System.out.println("Limit = " + buffer.limit());
        System.out.println("Position = " + buffer.position());
        System.out.println("Remaining = " + buffer.remaining());

        buffer.put((byte) 10).put((byte) 20).put((byte) 30);

        System.out.println("Capacity = " + buffer.capacity());
        System.out.println("Limit = " + buffer.limit());
        System.out.println("Position = " + buffer.position());
        System.out.println("Remaining = " + buffer.remaining());

        for (int i = 0; i < buffer.position(); i++)
            System.out.println(buffer.get(i));
    }
}

Compile Listing 13-3 (javac BufferDemo.java), and run this application (java BufferDemo). You should observe the following output:

| Capacity = 7  |
| Limit = 7     |
| Position = 0  |
| Remaining = 7 |
| Capacity = 7  |
| Limit = 7     |
| Position = 3  |
| Remaining = 4 |
| 10            |
| 20            |
| 30            |

Figure 13-2 illustrates the state of the buffer following the three put() method calls and presented in the previous output.
Tip: For maximum efficiency, you can perform bulk data transfers by using the `ByteBuffer put(byte[] src)`, `ByteBuffer put(byte[] src, int offset, int length)`, `ByteBuffer get(byte[] dst)`, and `ByteBuffer get(byte[] dst, int offset, int length)` methods to write and read an array of bytes.

Flipping Buffers

After filling a buffer, you must prepare it for draining by a channel. When you pass the buffer as is, the channel accesses undefined data beyond the current position.

To solve this problem, you could reset the position to 0, but how would the channel know when the end of the inserted data had been reached? The solution is to work with the limit property, which indicates the end of the active portion of the buffer. Basically, you set the limit to the current position and then reset the current position to 0.

You could accomplish this task by executing the following code, which also clears any defined mark:

```java
buffer.limit(buffer.position()).position(0);
```

However, there’s an easier way to accomplish the same task, as shown here:

```java
buffer.flip();
```

In either case, the buffer is ready to be drained.

Assuming that `buffer.flip();` is executed at the end of Listing 13-3’s `main()` method, Figure 13-3 reveals what the buffer state would look like after calling `flip()`.
A call to buffer.remaining() would return 3. This value indicates the number of bytes available for draining (10, 20, and 30).

Listing 13-4 provides another buffer flipping demonstration, which uses a character buffer.

Listing 13-4. Writing Characters to and Reading Them from a Character Buffer

```java
import java.nio.CharBuffer;

public class BufferDemo {
    public static void main(String[] args) {
        String[] poem = {
            "Roses are red",
            "Violets are blue",
            "Sugar is sweet",
            "And so are you."
        };

        CharBuffer buffer = CharBuffer.allocate(50);

        for (int i = 0; i < poem.length; i++) {
            // Fill the buffer.
            for (int j = 0; j < poem[i].length(); j++)
                buffer.put(poem[i].charAt(j));

            // Flip the buffer so that its contents can be read.
            buffer.flip();

            // Drain the buffer.
            while (buffer.hasRemaining())
                System.out.print(buffer.get());
        }
    }
}
```

**Figure 13-3. The buffer is ready to be drained**
// Empty the buffer to prevent BufferOverflowException.
buffer.clear();

System.out.println();
}
}
}

Compile Listing 13-4 (javac BufferDemo.java), and run this application (java BufferDemo). You should observe the following output:

Roses are red
Violets are blue
Sugar is sweet
And so are you.

**Note**  
rewind() is similar to flip() but ignores the limit. Also, calling flip() twice doesn’t return you to the original state. Instead, the buffer has a zero size. Calling a put() method results in BufferOverflowException, and calling a get() method results in BufferUnderflowException or (in the case of get(int)), IndexOutOfBoundsException.

**Marking Buffers**

You can mark a buffer by invoking the mark() method and later return to the marked position by invoking reset(). For example, suppose you’ve executed ByteBuffer buffer = ByteBuffer.allocate(7);, followed by buffer.put((byte) 10).put((byte) 20).put((byte) 30).put((byte) 40);, followed by buffer.limit(4);. The current position and limit are set to 4.

Continuing, suppose you execute buffer.position(1).mark().position(3);. Figure 13-4 reveals the buffer state at this point.

![Buffer State Diagram]

*Figure 13-4. The mark has been set to position 1*
If you sent this buffer to a channel, byte 40 would be sent (the current position is 3 because of position(3)) and the position would advance to 4. If you subsequently executed buffer.reset(); and sent this buffer to the channel, the position would be set to the mark (1) and bytes 20, 30, and 40 (all bytes from the current position to one position below the limit) would be sent to the channel (and in that order).

Listing 13-5 demonstrates this mark/reset scenario.

Listing 13-5. Marking the Current Buffer Position and Resetting the Current Position to the Marked Position

```java
import java.nio.ByteBuffer;

public class BufferDemo {
    public static void main(String[] args) {
        ByteBuffer buffer = ByteBuffer.allocate(7);
        buffer.put((byte) 10).put((byte) 20).put((byte) 30).put((byte) 40);
        buffer.limit(4);
        buffer.position(1).mark().position(3);
        System.out.println(buffer.get());
        System.out.println();
        buffer.reset();
        while (buffer.hasRemaining())
            System.out.println(buffer.get());
    }
}
```

Compile Listing 13-5 (javac BufferDemo.java), and run this application (java BufferDemo). You should observe the following output:

40
20
30
40

**Caution** Don’t confuse reset() with clear(). The clear() method marks a buffer as empty whereas reset() changes the buffer’s current position to the previously set mark, or throws InvalidMarkException when there’s no previously set mark.
**Buffer Subclass Operations**

ByteBuffer and the other primitive-type buffer classes declare a compact() method that’s useful for compacting a buffer by copying all bytes between the current position and the limit to the beginning of the buffer. The byte at index \( p = \text{position()} \) is copied to index 0, the byte at index \( p + 1 \) is copied to index 1, and so on until the byte at index \( \text{limit()} - 1 \) is copied to index \( n = \text{limit()} - 1 - p \). The buffer's current position is then set to \( n + 1 \) and its limit is set to its capacity. The mark, when defined, is discarded.

You invoke compact() after writing data from a buffer to handle situations where not all of the buffer’s content is written. Consider the following example that copies content from an in channel to an out channel via buffer buf:

```java
buf.clear(); // Prepare buffer for use
while (in.read(buf) != -1)
{
    buf.flip();  // Prepare buffer for draining.
    out.write(buf); // Write the buffer.
    buf.compact(); // Do this in case of a partial write.
}
```

The compact() method call moves unwritten buffer data to the beginning of the buffer so that the next read() method call appends read data to the buffer’s data instead of overwriting that data when compact() isn’t specified.

You may occasionally need to compare buffers for equality or order. All Buffer subclasses except for ByteBuffer's MappedByteBuffer subclass override the equals() and compareTo() methods to perform these comparisons—MappedByteBuffer inherits these methods from its ByteBuffer superclass. The following example shows you how to compare byte buffers `bytBuf1` and `bytBuf2` for equality and ordering:

```java
System.out.println(bytBuf1.equals(bytBuf2));
System.out.println(bytBuf1.compareTo(bytBuf2));
```

The equals() contract for ByteBuffer states that 2 byte buffers are equal if and only if they have the same element type; they have the same number of remaining elements; and the two sequences of remaining elements, considered independently of their starting positions, are individually equal. This contract is the same for the other Buffer subclasses.

The compareTo() method for ByteBuffer states that 2 byte buffers are compared for order by comparing their sequences of remaining elements lexicographically, without regard to the starting position of each sequence within its corresponding buffer. Pairs of byte elements are compared as if by invoking Byte.compare(byte, byte). Similar descriptions apply to the other Buffer subclasses.

**Byte Ordering**

Nonbyte primitive types, except for Boolean (which might be represented by a bit or by a byte), are composed of several bytes: a character or a short integer occupies 2 bytes, a 32-bit integer or a floating-point value occupies 4 bytes, and a long integer or a double precision floating-point value occupies 8 bytes. Each value of one of these multibyte types is stored in a sequence of contiguous memory locations. However, the order of these bytes can differ from platform to platform.
For example, consider 32-bit long integer 0x10203040. This value’s 4 bytes could be stored in memory (from low address to high address) as 10, 20, 30, 40; this arrangement is known as **big-endian order** (the most significant byte, the “big” end, is stored at the lowest address). Alternatively, these bytes could be stored as 40, 30, 20, 10; this arrangement is known as **little-endian order** (the least significant byte, the “little” end, is stored at the lowest address).

Java provides the [java.nio.ByteOrder](https://docs.oracle.com/javase/8/docs/api/) class to help you deal with byte-order issues when writing/reading multibyte values to/from a multibyte buffer. `ByteOrder` declares a `ByteOrder nativeOrder()` method that returns the platform’s byte order as a `ByteOrder` instance. Because this instance is one of `ByteOrder`’s `BIG_ENDIAN` and `LITTLE_ENDIAN` constants, and because no other `ByteOrder` instances can be created, you can compare `nativeOrder()`’s return value to one of these constants via the `==` or `!=` operator.

Also, each multibyte class (such as `FloatBuffer`) declares a `ByteOrder order()` method that returns the buffer’s byte order. This method returns `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN`.

The `ByteOrder` value returned from `order()` can take on a different value based on how the buffer was created. If a multibyte buffer (such as a float buffer) was created by allocation or by wrapping an existing array, the buffer’s byte order is the native order of the underlying platform. However, if a multibyte buffer was created as a view of a byte buffer, the view buffer’s byte order is that of the byte buffer when the view was created. The view buffer’s byte order cannot be subsequently changed.

`ByteBuffer` differs from the multibyte classes when it comes to byte order. Its default byte order is always big endian, even when the underlying platform’s byte order is little endian. `ByteBuffer` defaults to big endian because Java’s default byte order is also big endian, which lets classfiles and serialized objects store data consistently across virtual machines.

Because this big endian default can impact performance on little-endian platforms, `ByteBuffer` also declares a `ByteBuffer order(ByteOrder bo)` method to change the byte buffer’s byte order.

Although it may seem unusual to change the byte order of a byte buffer (where only single-byte data items are accessed), this method is useful because `ByteBuffer` also declares several convenience methods for writing and reading multibyte values (`ByteBuffer putInt(int value)` and `int getInt()`, for example). These convenience methods write these values according to the byte buffer’s current byte order. Furthermore, you can subsequently call `ByteBuffer`’s `LongBuffer asLongBuffer()` or another `asXBuffer()` method to return a view buffer whose order will reflect the byte buffer’s changed byte order.

### Direct Byte Buffers

Unlike multibyte buffers, byte buffers can serve as the sources and/or targets of channel-based I/O. This shouldn’t come as a surprise because operating systems perform I/O on memory areas that are contiguous sequences of 8-bit bytes (not floating-point values, not 32-bit integers, and so on).

Operating systems can directly access the address space of a process. For example, an operating system could directly access a virtual machine process’s address space to perform a data transfer operation based on a byte array. However, a virtual machine might not store the array of bytes contiguously or its garbage collector might move the byte array to another location. Because of these limitations, direct byte buffers were created.
A **direct byte buffer** is a byte buffer that interacts with channels and native code to perform I/O. The direct byte buffer attempts to store byte elements in a memory area that a channel uses to perform direct (raw) access via native code that tells the operating system to drain or fill the memory area directly.

Direct byte buffers are the most efficient means for performing I/O on the virtual machine. Although you can also pass nondirect byte buffers to channels, a performance problem might arise because nondirect byte buffers are not always able to serve as the target of native I/O operations.

When passed a nondirect byte buffer, a channel might have to create a temporary direct byte buffer, copy the nondirect byte buffer’s content to the direct byte buffer, perform the I/O operation on the temporary direct byte buffer, and copy the temporary direct byte buffer’s content to the nondirect byte buffer. The temporary direct byte buffer will then be subject to garbage collection.

Although optimal for I/O, a direct byte buffer can be expensive to create because memory extraneous to the virtual machine’s heap will need to be allocated by the operating system, and setting up/tearing down this memory might take longer than when the buffer was located within the heap. After your code is working and should you want to experiment with performance optimization, you can easily obtain a direct byte buffer by invoking ByteBuffer’s allocateDirect() method, which I discussed earlier.

### Working with Channels

Channels partner with buffers to achieve high-performance I/O. A **channel** is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that’s capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations.

> **Note** Channels are the gateways through which native I/O services are accessed. Channels use byte buffers as the endpoints for sending and receiving data.

There often exists a one-to-one correspondence between an operating system file handle or file descriptor and a channel. When you work with channels in a file context, the channel will often be connected to an open file descriptor. Despite channels being more abstract than file descriptors, they are still capable of modeling an operating system’s native I/O facilities.

### Channel and Its Children

Java supports channels via its java.nio.channels and java.nio.channels.spi packages. Applications interact with the types located in the former package; developers who are defining new selector providers work with the latter package. (I will discuss selectors later in this chapter.)
All channels are instances of classes that ultimately implement the `java.nio.channels.Channel` interface. `Channel` declares the following methods:

- **void close()**: Closes this channel. When this channel is already closed, invoking close() has no effect. When another thread has already invoked close(), a new close() invocation blocks until the first invocation finishes, after which close() returns without effect. This method throws `IOException` when an I/O error occurs. After the channel is closed, any further attempts to invoke I/O operations upon it result in `java.nio.channels.ClosedChannelException` being thrown.

- **boolean isOpen()**: Returns this channel's open status. This method returns true when the channel is open; otherwise, it returns false.

These methods indicate that only two operations are common to all channels: close the channel and determine whether the channel is open or closed. To support I/O, `Channel` is extended by the `java.nio.channels.WritableByteChannel` and `java.nio.channels.ReadableByteChannel` interfaces:

- **WritableByteChannel** declares an abstract int `write(ByteBuffer buffer)` method that writes a sequence of bytes from buffer to the current channel. This method returns the number of bytes actually written. It throws `java.nio.channels.NonWritableChannelException` when the channel was not opened for writing, `java.nio.channels.ClosedChannelException` when the channel is closed, `java.nio.channels.AsynchronousCloseException` when another thread closes the channel during the write, `java.nio.channels.ClosedByInterruptException` when another thread interrupts the current thread while the write operation is in progress (thereby closing the channel and setting the current thread’s interrupt status), and `java.io.IOException` when some other I/O error occurs.

- **ReadableByteChannel** declares an abstract int `read(ByteBuffer buffer)` method that reads bytes from the current channel into buffer. This method returns the number of bytes actually read (or -1 when there are no more bytes to read). It throws `java.nio.channels.NonReadableChannelException` when the channel was not opened for reading; `ClosedChannelException` when the channel is closed; `AsynchronousCloseException` when another thread closes the channel during the read; `ClosedByInterruptException` when another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread's interrupt status; and `IOException` when some other I/O error occurs.
CHAPTER 13: Migrating to New I/O

Note  A channel whose class implements only WritableByteChannel or ReadableByteChannel is unidirectional. Attempting to read from a writable byte channel or write to a readable byte channel results in a thrown exception.

You can use the instanceof operator to determine if a channel instance implements either interface. Because it's somewhat awkward to test for both interfaces, Java supplies the java.nio.channels.ByteChannel interface, which is an empty marker interface that subtypes WritableByteChannel and ReadableByteChannel. When you need to learn whether or not a channel is bidirectional, it's more convenient to specify an expression such as channel instanceof ByteChannel.

Channel is also extended by the java.nio.channels.InterruptibleChannel interface. InterruptibleChannel describes a channel that can be asynchronously closed and interrupted. This interface overrides its Channel superinterface's close() method header, presenting the following additional stipulation to Channel's contract for this method: any thread currently blocked in an I/O operation upon this channel will receive AsynchronousCloseException (an IOException descendent).

A channel that implements this interface is asynchronously closeable: When a thread is blocked in an I/O operation on an interruptible channel, another thread may invoke the channel's close() method. This causes the blocked thread to receive a thrown AsynchronousCloseException instance.

A channel that implements this interface is also interruptible: when a thread is blocked in an I/O operation on an interruptible channel, another thread may invoke the blocked thread's interrupt() method. Doing this causes the channel to be closed, the blocked thread to receive a thrown ClosedByInterruptException instance, and the blocked thread to have its interrupt status set. (When a thread’s interrupt status is already set and it invokes a blocking I/O operation on a channel, the channel is closed and the thread will immediately receive a thrown ClosedByInterruptException instance; its interrupt status will remain set.)

NIO's designers chose to shut down a channel when a blocked thread is interrupted because they couldn't find a way to handle interrupted I/O operations reliably in the same manner across platforms. The only way to guarantee deterministic behavior was to shut down the channel.

Tip  You can determine whether or not a channel supports asynchronous closing and interruption by using the instanceof operator in an expression such as channel instanceof InterruptibleChannel.
CHAPTER 13: Migrating to New I/O

You previously learned that you must call a class method on a Buffer subclass to obtain a buffer. Regarding channels, there are two ways to obtain a channel:

- The java.nio.channels package provides a Channels utility class that offers two methods for obtaining channels from streams—for each of the following methods, the underlying stream is closed when the channel is closed, and the channel isn’t buffered:
  - WritableByteChannel newChannel(OutputStream outputStream) returns a writable byte channel for the given OutputStream.
  -ReadableByteChannel newChannel(InputStream inputStream) returns a readable byte channel for the given InputStream.

- Various classic I/O classes have been retrofitted to support channel creation. For example, RandomAccessFile declares a FileChannel getChannel() method for returning a file channel instance, and java.net.Socket declares a SocketChannel getChannel() method for returning a socket channel.

Listing 13-6 uses the Channels class to obtain channels for the standard input and output streams and then uses these channels to copy bytes from the input channel to the output channel.

**Listing 13-6. Copying Bytes from an Input Channel to an Output Channel**

```java
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.Channels;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;

public class ChannelDemo {
    public static void main(String[] args) {
        ReadableByteChannel src = Channels.newChannel(System.in);
        WritableByteChannel dest = Channels.newChannel(System.out);

        try {
            copy(src, dest); // or copyAlt(src, dest);
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        } finally {
            try {
                src.close();
                dest.close();
            }
        }
    }
}
```
catch (IOException ioe)
{
    ioe.printStackTrace();
}
}

static void copy(ReadableByteChannel src, WritableByteChannel dest)
    throws IOException
{
    ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
    while (src.read(buffer) != -1)
    {
        buffer.flip();
        dest.write(buffer);
        buffer.compact();
    }
    buffer.flip();
    while (buffer.hasRemaining())
        dest.write(buffer);
}

static void copyAlt(ReadableByteChannel src, WritableByteChannel dest)
    throws IOException
{
    ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
    while (src.read(buffer) != -1)
    {
        buffer.flip();
        while (buffer.hasRemaining())
            dest.write(buffer);
        buffer.clear();
    }
}

Listing 13-6 presents two approaches to copying bytes from the standard input stream to the standard output stream. In the first approach, which is exemplified by the copy() method, the goal is to minimize native I/O calls (via the write() method calls), although more data may end up being copied as a result of the compact() method calls. In the second approach, as demonstrated by copyAlt(), the goal is to eliminate data copying, although more native I/O calls might occur.

Each of copy() and copyAlt() first allocates a direct byte buffer (recall that a direct byte buffer is the most efficient means for performing I/O on the virtual machine) and enters a while loop that continually reads bytes from the source channel until end-of-input (read() returns -1). Following the read, the buffer is flipped so that it can be drained. Here is where the methods diverge.
The copy() method while loop makes a single call to write(). Because write() might not completely drain the buffer, compact() is called to compact the buffer before the next read. Compaction ensures that unwritten buffer content isn't overwritten during the next read operation. Following the while loop, copy() flips the buffer in preparation for draining any remaining content and then works with hasRemaining() and write() to drain the buffer completely.

The copyAlt() method while loop contains a nested while loop that works with hasRemaining() and write() to continue draining the buffer until the buffer is empty. This is followed by a clear() method call that empties the buffer so that it can be filled on the next read() call.

Note: It's important to realize that a single write() method call may not output the entire content of a buffer. Similarly, a single read() call may not completely fill a buffer.

Compile Listing 13-6 via the following command line:

```
javac ChannelDemo.java
```

Run ChannelDemo via the following command lines:

```
java ChannelDemo
java ChannelDemo <ChannelDemo.java >ChannelDemo.bak
```

The first command line copies standard input to standard output. The second command line copies the contents of ChannelDemo.java to ChannelDemo.bak. After testing the copy() method, replace copy(src, dest); with copyAlt(src, dest); and repeat.

**Channels in Depth**

The previous discussion of the Channel interface and its direct descendents has given you some insight into NIO channels. However, there's much more to explore. This section takes you deeper into channels by exploring scatter/gather I/O, file channels, socket channels, and pipes.

**Scatter/Gather I/O**

Channels provide the ability to perform a single I/O operation across multiple buffers. This capability is known as scatter/gather I/O (and is also known as vectored I/O).

In the context of a write operation, the contents of several buffers are gathered (drained) in sequence and then sent through the channel to a destination; these buffers are not required to have identical capacities. In the context of a read operation, the contents of a channel are scattered (filled) to multiple buffers in sequence; each buffer is filled to its limit until the channel is empty or total buffer space is used up.
Note  Modern operating systems provide APIs that support vectored I/O to eliminate (or at least reduce) system calls or buffer copies, and hence improve performance. For example, the Win32/Win64 APIs provide ReadFileScatter() and WriteFileGather() functions for this purpose.

Java provides the java.nio.channels.ScatteringByteChannel interface to support scattering and the java.nio.channels.GatheringByteChannel interface to support gathering.

ScatteringByteChannel offers the following methods:
- long read(ByteBuffer[] buffers, int offset, int length)
- long read(ByteBuffer[] buffers)

GatheringByteChannel offers the following methods:
- long write(ByteBuffer[] buffers, int offset, int length)
- long write(ByteBuffer[] buffers)

The first read() method and the first write() method let you identify the first buffer to read/write by passing a zero-based offset to offset, and the number of buffers to read/write by passing a value to length. The second read() method and the second write() method read and write all buffers in sequence.

Listing 13-7 demonstrates read(ByteBuffer[] buffers) and write(ByteBuffer[] buffers).

Listing 13-7. Demonstrating Scatter/Gather

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.Channels;
import java.nio.channels.GatheringByteChannel;
import java.nio.channels.ScatteringByteChannel;

public class ChannelDemo
{
    public static void main(String[] args) throws IOException
    {
        ScatteringByteChannel src;
        src = (ScatteringByteChannel) Channels.newChannel(new FileInputStream("x.dat"));
        ByteBuffer buffer1 = ByteBuffer.allocateDirect(5);
        ByteBuffer buffer2 = ByteBuffer.allocateDirect(3);
        ByteBuffer[] buffers = { buffer1, buffer2 };
        src.read(buffers);
        buffer1.flip();
```
while (buffer1.hasRemaining())
    System.out.println(buffer1.get());
System.out.println();
buffer2.flip();
while (buffer2.hasRemaining())
    System.out.println(buffer2.get());
buffer1.rewind();
buffer2.rewind();
GatheringByteChannel dest;
dest = (GatheringByteChannel) Channels.newChannel(new FileOutputStream("y.dat");
buffers[0] = buffer2;
buffers[1] = buffer1;
dest.write(buffers);
}

Listing 13-7's main() method first obtains a scattering byte channel by instantiating java.io.
FileInputStream and passing this instance to the Channels class's ReadableByteChannel
newChannel(InputStream inputStream) method. The returned ReadableByteChannel instance is
cast to ScatteringByteChannel because this instance is actually a file channel (discussed later) that
implements ScatteringByteChannel.

Next, main() creates a couple of direct byte buffers; the first buffer has a capacity of 5 bytes and the
second buffer has a capacity of 3 bytes. These buffers are subsequently stored in an array, and this
array is passed to read(ByteBuffer[]) to fill them.

After filling the buffers, main() flips them so that it can output their contents to standard output. After
these contents have been output, the buffers are rewound in preparation for being drained via a
gather operation.

main() now obtains a gathering byte channel by instantiating java.io.FileOutputStream and
passing this instance to the Channels class's WritableByteChannel newChannel(OutputStream
outputStream) method. The returned WritableByteChannel instance is cast to GatheringByteChannel
because this instance is actually a file channel (discussed later) that implements
GatheringByteChannel.

Finally, main() assigns these buffers to the buffers array in reverse order to how they were originally
assigned, and then passes this array to write(ByteBuffer[]) to drain them.

Create a file named x.dat, and store the following text in this file:

12345abcdefg
Now compile Listing 13-7 (javac ChannelDemo.java), and run this application (java ChannelDemo). You should observe the following Unicode values for the first eight characters:

49 50 51 52 53 97 98 99

Additionally, you should observe a newly created y.dat file with the following content:

abc12345

**File Channels**

I previously mentioned that `RandomAccessFile` declares a `FileChannel getChannel()` method for returning a file channel instance, which describes an open connection to a file. It turns out that `FileInputStream` and `FileOutputStream` also provide the same method. In contrast, `FileReader` and `FileWriter` don't offer a way to obtain a file channel.

**Caution** The file channel returned from `FileInputStream`'s `getChannel()` method is read-only, and the file channel returned from `FileOutputStream`'s `getChannel()` method is write-only. Attempting to write to a read-only file channel or read from a write-only file channel results in an exception.

The abstract `java.nio.channels.FileChannel` class describes a file channel. Because this class implements the `InterruptibleChannel` interface, file channels are interruptible. Because this class implements the `ByteChannel`, `GatheringByteChannel`, and `ScatteringByteChannel` interfaces, you can write to, read from, and perform scattering I/O on underlying files. However, there's more.

**Note** Unlike buffers, which are not thread-safe, file channels are thread-safe.

A file channel maintains a current position into the file, which `FileChannel` lets you obtain and change. It also lets you request that cached data be forced to the disk, read/write file content, obtain the size of the file underlying the channel, truncate a file, attempt to lock the entire file or just a region of the file, perform memory-mapped file I/O, and transfer data directly to another channel in a manner that has the potential to be optimized by the platform.

Table 13-2 describes a few of `FileChannel`'s methods.
Table 13-2. FileChannel Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void force(boolean metadata)</td>
<td>Requests that all updates to this file channel be committed to the storage device. When this method returns, all modifications made to the platform file underlying this channel have been committed when the file resides on a local storage device. However, when the file isn’t hosted locally (it’s on a networked file system, for example), applications cannot be certain that the modifications have been committed. (No assurances are given that changes made to the file using methods defined elsewhere will be committed. For example, changes made via a mapped byte buffer may not be committed.) The metadata value indicates whether the update should include the file’s metadata (such as last modification time and last access time), when true is passed, or not include the file’s metadata, when false is passed. Passing true may invoke an underlying write to the operating system (if the platform is maintaining metadata, such as last access time), even when the channel is opened as a read-only channel. This method throws ClosedChannelException when the channel is already closed and throws IOException when any other I/O error occurs.</td>
</tr>
<tr>
<td>long position()</td>
<td>Returns the current zero-based file position maintained by this file channel. This method throws ClosedChannelException when the file channel is closed and IOException when another I/O error occurs.</td>
</tr>
<tr>
<td>FileChannel position(long newPosition)</td>
<td>Sets this file channel’s current file position to newPosition. The argument is the number of bytes counted from the start of the file. The position cannot be set to a negative value. However, it can be set beyond the current file size. If set beyond the current file size, attempts to read will return end of file. Write operations will succeed, but they will fill the bytes between the current end of file and the new position with the required number of (unspecified) byte values. This method throws IllegalArgumentException when offset is negative, ClosedChannelException when the file channel is closed, and IOException when another I/O error occurs.</td>
</tr>
<tr>
<td>int read(ByteBuffer buffer)</td>
<td>Reads bytes from this file channel into the given buffer. The maximum number of bytes that will be read is the remaining number of bytes in the buffer when the method is invoked. The bytes will be copied into the buffer starting at the buffer’s current position. The call may block when other threads are also attempting to read from this channel. Upon completion, the buffer’s position is set to the end of the bytes that have been read. The buffer’s limit isn’t changed. This method returns the number of bytes actually read and throws the same exceptions as previously discussed regarding ReadableByteChannel.</td>
</tr>
</tbody>
</table>
The **force()** method ensures that all changes made to a file residing in the local filesystem, and since this method was previously invoked, are written to the disk. This capability is vital for critical tasks such as transaction processing where you must maintain data integrity and ensure reliable recovery. However, this guarantee doesn’t apply to remote filesystems.

Passing `true` to `force()` results in metadata (last modification time, access permissions, and so on) also being synchronized to the disk. Because metadata isn’t usually critical to file recovery, you can often pass `false` and gain a small performance increase because an extra I/O operation isn’t required to output the metadata.

FileChannel objects support the concept of a current file position, which determines the location where the next data item will be read from or written to. The `position()` method returns the current position, and the `position(long newPosition)` method sets the current position to `newPosition`. The value passed to `newPosition` must be nonnegative or `IllegalArgumentException` will be thrown.

The are two forms of the `read()` and `write()` methods. The relative forms don’t take position arguments, and they ensure that the current file position is updated after a call to either method. The absolute forms of these methods take a position argument and don’t update the position. Absolute reads and writes can be more efficient because the channel’s state doesn’t need to be updated.

### Table 13-2. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int read(ByteBuffer dst, long position)</code></td>
<td>Equivalent to the previous method except that bytes are read starting at the specified file position. <code>IllegalArgumentException</code> is thrown when position is negative.</td>
</tr>
<tr>
<td><code>long size()</code></td>
<td>Returns the size (in bytes) of the file underlying this file channel. This method throws <code>ClosedChannelException</code> when the file channel is closed and <code>IOException</code> when another I/O error occurs.</td>
</tr>
<tr>
<td><code>FileChannel truncate(long size)</code></td>
<td>Truncates the file underlying this file channel to size. Any bytes beyond the given size are removed from the file. When there are no bytes beyond the given size, the file contents are unmodified. When the current file position is greater than the given size, it’s set to size.</td>
</tr>
<tr>
<td><code>int write(ByteBuffer buffer)</code></td>
<td>Writes a sequence of bytes to this file channel from the given buffer. Bytes are written starting at the channel’s current file position unless the channel is in append mode, in which case the position is first advanced to the end of the file. The file is grown (when necessary) to accommodate the written bytes, and then the file position is updated with the number of bytes actually written. Otherwise, this method behaves exactly as specified by the <code>WritableByteChannel</code> interface. This method returns the number of bytes actually written and throws the same exceptions as previously discussed regarding <code>WritableByteChannel</code>.</td>
</tr>
<tr>
<td><code>int write(ByteBuffer src, long position)</code></td>
<td>Equivalent to the previous method except that bytes are written starting at the specified file position. <code>IllegalArgumentException</code> is thrown when position is negative.</td>
</tr>
</tbody>
</table>
If you attempt to perform an absolute read past the end of a file, which \texttt{size()} returns, -1 is returned to signify end of file. Attempting to perform an absolute write past the end of a file causes the file to grow to accommodate the bytes being written. The values of the bytes located between the previous end of file and the first newly written byte are filesystem-specific and may constitute a hole.

A \textit{hole} occurs in a file when the amount of disk space allocated for the file is smaller than the file’s size. Modern filesystems typically allocate space only for data that’s written to the file. When data is written to noncontiguous areas, holes can appear. When the file is read, holes typically appear to be zero-filled but don’t take up disk space.

The \texttt{truncate(long size)} method is useful for reducing a file’s size. This method truncates all data beyond the specified size. When the file’s size is greater than the specified size, all bytes past the specified size are discarded. When the specified size is greater than or equal to the current size, the file isn’t changed.

Listing 13-8 demonstrates various methods from Table 13-2.

\textit{Listing 13-8. Demonstrating a File Channel}

\begin{verbatim}
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class ChannelDemo {
    public static void main(String[] args) throws IOException {
        RandomAccessFile raf = new RandomAccessFile("temp", "rw");
        FileChannel fc = raf.getChannel();
        long pos;
        System.out.println("Position = " + (pos = fc.position()));
        System.out.println("size: " + fc.size());
        String msg = "This is a test message."
        ByteBuffer buffer = ByteBuffer.allocateDirect(msg.length() * 2);
        buffer.asCharBuffer().put(msg);
        fc.write(buffer);
        fc.force(true);
        System.out.println("position: " + fc.position());
        System.out.println("size: " + fc.size());
        buffer.clear();
        fc.position(pos);
        fc.read(buffer);
        buffer.flip();
        while (buffer.hasRemaining())
            System.out.print(buffer.getChar());
    }
}
\end{verbatim}

www.it-ebooks.info
Listing 13-8’s `main()` method first creates a randomly-accessible file named `temp` for writing and reading. It then obtains a channel for communicating with this file and reports the file channel’s current position and the file’s size, which are both 0 for a newly created file.

`main()` next allocates a direct byte buffer for storing a message to be written to the file, treats this buffer as a character buffer, and calls the character buffer’s `put()` method to store the message in the buffer, which is then output to the file.

`main()` now calls `force(true)` to recommend to the underlying operating system that the data be committed to the underlying storage device.

After reporting the new current position and file size, `main()` clears the buffer, resets the current file position to the position that was current before the message was written, and reads the previously written content back into the buffer. It then flips the buffer and outputs its contents.

Compile Listing 13-8 (`javac ChannelDemo.java`), and run this application (`java ChannelDemo`). You should observe the following output:

```
Position = 0
size: 46
position: 46
size: 46
This is a test message.
```

### Locking Files

The ability to lock all or part of a file was an important but missing feature in Java until Java 1.4 arrived. This capability lets a virtual machine process prevent other processes from accessing all or part of a file until it’s finished with the entire file or part of the file.

**Note** Think of a `process` as an executing application, such as the `java` command which loads and starts running the Java virtual machine.

Although an entire file can be locked, it’s often desirable to lock a smaller region. For example, a database management system might lock individual table rows that are being updated instead of locking the entire table so that read requests can be honored, which improves throughput.

Locks that are associated with files are known as `file locks`. Each file lock starts at a certain byte position in the file and has a specific length (in bytes) from this position. Together, they define the region governed by the lock. File locks let processes coordinate access to various regions in a file.

There are two kinds of file locks: exclusive and shared. An `exclusive lock` prevents other file locks from being used within the region governed by the exclusive lock. In contrast, a `shared lock` may apply to a region governed by other shared locks.

Exclusive and shared locks are commonly used in scenarios where a shared file is primarily read and occasionally updated. A process that needs to read from the file acquires a shared lock to the entire
file or to the desired subregion. A second process that also needs to read from the file acquires a
shared lock to the desired region. Both processes can read the file without interfering with each other.

Suppose a third process wants to perform updates. To do so, it would request an exclusive lock. The
process would block until all exclusive or shared locks that overlap with its region were released.
Once the exclusive lock was granted to the updater process, any reader process requesting a
shared lock would block until the exclusive lock was released. The updater process could then
update the file without the reader processes observing inconsistent data.

There are a couple more items to keep in mind regarding file locking:

- When an operating system doesn’t support shared locks, a shared lock request
  is quietly promoted to a request for an exclusive lock. Although correctness is
  assured, performance may be impacted.
- Locks are applied on a per-file basis. They are not applied on a per-thread or
  per-process basis. Two threads running on the same virtual machine that
  request an exclusive lock to the same region will be granted access.

FileChannel declares four methods for obtaining exclusive and shared locks:

- FileLock lock(): Obtains an exclusive lock on this file channel's underlying
  file. This convenience method is equivalent to executing fileChannel.lock(0L,
  Long.MAX_VALUE, false); where fileChannel references a file channel.
  This method returns a java.nio.channels.FileLock object representing the
  locked area. It throws ClosedChannelException when the file channel is closed;
  NonWritableChannelException when the channel isn’t open for writing; java.nio.
  channels.OverlappingFileLockException when either a lock is already held that
  overlaps this lock request or another thread is waiting to acquire a lock that will
  overlap with this request; java.nio.channels.FileLock InterruptionException
  when the calling thread was interrupted while waiting to acquire the lock;
  AsynchronousCloseException when the channel was closed while the calling
  thread was waiting to acquire the lock; and IOException when some other I/O
  error occurs while obtaining the requested lock.
- FileLock lock(long position, long size, boolean shared): This method is
  similar to the previous method except that it attempts to acquire a lock on the
given region of this channel's file. Pass nonnegative values to position and size
to delimit the region. Pass true to shared to request a shared lock and false to
shared to request an exclusive lock.
- FileLock tryLock(): Attempts to obtain an exclusive lock on this file channel's
  underlying file without blocking. This convenience method is equivalent to
  executing fileChannel.tryLock(0L, Long.MAX_VALUE, false); where
  fileChannel references a file channel.
  This method returns a FileLock object representing the locked area or null when
  the lock would overlap with an existing exclusive lock in another operating system
  process. It throws ClosedChannelException when the file channel is closed;
  OverlappingFileLockException when a lock that overlaps the requested region is
already held by this virtual machine, or when another thread is already blocked in this method and is attempting to lock an overlapping region; and IOException when some other I/O error occurs while obtaining the requested lock.

- **FileLock tryLock(long position, long size, boolean shared):** This method is similar to the previous method except that it attempts to acquire a lock on the given region of this channel's file. Pass nonnegative values to `position` and `size` to delimit the region. Pass `true` to `shared` to request a shared lock and `false` to `shared` to request an exclusive lock.

The `lock()` methods block when the desired region to be locked is already locked (unless both locks are shared locks). In contrast, the `tryLock()` methods return immediately with a null value.

Each method returns a `FileLock` instance, which encapsulates a locked region in the file. `FileLock`'s methods are as follows:

- **FileChannel channel():** Returns the file channel on whose file this lock was acquired or null when the lock wasn't acquired by a file channel.
- **void close():** Invokes the `release()` method to release the lock.
- **boolean isShared():** Returns `true` to identify the lock as a shared lock or `false` to identify the lock as an exclusive lock.
- **boolean isValid():** Returns `true` to identify a valid lock; otherwise, returns `false`. A lock is valid until it's released or the associated file channel is closed, whichever comes first.
- **boolean overlaps(long position, long size):** Indicates whether (return `true`) or not (return `false`) this lock's region overlaps the region described in the parameter list.
- **long position():** Returns the position within the file of the first byte of the locked region. A locked region doesn't need to be contained within or even overlap the underlying file, so the value returned by this method may exceed the file's current size.
- **void release():** Releases this lock. If this lock object is valid, invoking this method releases the lock and renders the object invalid. If this lock object is invalid, invoking this method has no effect.
- **long size():** Returns the length of the file lock (in bytes).
- **String toString():** Returns a string describing the range, type, and validity of this lock.

A `FileLock` instance is associated with a `FileChannel` instance but the file lock represented by the `FileLock` instance associates with the underlying file and not with the file channel. Without care, you can run into conflicts (and possibly even a deadlock) when you don't release a file lock after you're
finished using it. To avoid these problems, you should adopt a pattern such as the following one in order to ensure that the file lock is always released:

```java
FileLock lock = fileChannel.lock();
try {
    // interact with the file channel
} catch (IOException ioe) {
    // handle the exception
} finally {
    lock.release();
}
```

I've created an application that demonstrates file locking. It follows this pattern to ensure that the lock is released. Listing 13-9 presents its source code.

**Listing 13-9. Demonstrating File Locking**

```java
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;

public class ChannelDemo {
    final static int MAXQUERIES = 150000;
    final static int MAXUPDATES = 150000;

    final static int RECLEN = 16;

    static ByteBuffer buffer = ByteBuffer.allocate(RECLEN);
    static IntBuffer intBuffer = buffer.asIntBuffer();

    static int counter = 1;

    public static void main(String[] args) throws IOException {
        boolean writer = false;
        if (args.length != 0)
            writer = true;
        RandomAccessFile raf = new RandomAccessFile("temp",
            (writer) ? "rw" : "r");
```
FileChannel fc = raf.getChannel();
if (writer)
    update(fc);
else
    query(fc);
}

static void query(FileChannel fc) throws IOException {
    for (int i = 0; i < MAXQUERIES; i++) {
        System.out.println("acquiring shared lock");
        FileLock lock = fc.lock(0, RECLEN, true);
        try {
            buffer.clear();
            fc.read(buffer, 0);
            int a = intBuffer.get(0);
            int b = intBuffer.get(1);
            int c = intBuffer.get(2);
            int d = intBuffer.get(3);
            System.out.println("Reading: "+ a + " " +
                                b + " " +
                                c + " " +
                                d);
            if (a*2 != b || a*3 != c || a*4 != d)
                System.out.println("error");
                return;
        }
        finally {
            lock.release();
        }
    }
    finally {
    
    }
}

static void update(FileChannel fc) throws IOException {
    for (int i = 0; i < MAXUPDATES; i++) {
        System.out.println("acquiring exclusive lock");
        FileLock lock = fc.lock(0, RECLEN, false);
        try {
            intBuffer.clear();
            int a = counter;
            int b = counter*2;
            int c = counter*3;
            int d = counter*4;
Listing 13-9 describes an application that either updates a file named temp or queries this file. Because file locking applies at the process level and not at the thread level, you need to run two copies of this application to demonstrate file locking for yourself. One copy will behave as a writer, updating the file. The other copy will behave as a reader, querying the file.

The ChannelDemo class first declares a pair of constants for controlling the duration of the update and query loops, along with a constant that denotes the length of a record. It then allocates a byte buffer that can accommodate the entire 16-byte record and an int-based view buffer for treating the byte buffer as a sequence of four int values. Finally, a counter variable initialized to 1 is declared.

The main() method first determines whether the application runs as a writer or runs as a reader. If you specify any command-line arguments, writer is assumed. This method then either opens (for a reader) or creates (for a writer) temp as a random access file. If this file doesn’t exist when you run the application as a reader, an exception is thrown.

After opening or creating temp, a file channel to this file is obtained. This channel is then passed to either the update() or query() method.

Consider update(). This method receives the file channel argument and enters a fixed-length for loop whose duration is governed by the MAXUPDATES constant. After outputting a lock-acquisition message, it attempts to obtain an exclusive lock to the entire 16-byte record. If the reader process has locked that record via a shared lock, lock() blocks until the shared lock is released.

Once the lock is obtained, the view buffer is cleared, which sets the position to 0 and the limit to the capacity. The buffer is ready to be completely filled.

The counter variable’s current value is now accessed and saved in a variable. This value is multiplied by 2, 3, and 4, and the results are also saved in their own variables. After outputting a writing message that identifies these values, update() makes four put() calls on the view buffer to store the values in the byte buffer. The counter variable is incremented and the byte buffer is cleared to ensure that it can be completely drained. Finally, the file channel’s write() method is called to drain the buffer to the underlying temp file.
The query() method has a similar structure to the update() method. However, it uses the file channel to read the temp file's record, and it stores the results in the byte buffer. After outputting a message to display the read results, it verifies that the values are correct. Any deviation from what is expected causes the method to terminate after outputting an error message.

Compile Listing 13-9 (javac ChannelDemo.java), and execute the following command line in one command window:

```
java ChannelDemo w
```

You should observe messages similar to the following:

```
acquiring exclusive lock
Writing: 1 2 3 4
acquiring exclusive lock
Writing: 2 4 6 8
acquiring exclusive lock
Writing: 3 6 9 12
acquiring exclusive lock
Writing: 4 8 12 16
acquiring exclusive lock
Writing: 5 10 15 20
```

In a second command window, execute the following command line:

```
java ChannelDemo
```

You should observe messages similar to the following:

```
acquiring shared lock
Reading: 2500 5000 7500 10000
acquiring shared lock
Reading: 2501 5002 7503 10004
acquiring shared lock
Reading: 2502 5004 7506 10008
acquiring shared lock
Reading: 2503 5006 7509 10012
acquiring shared lock
Reading: 2504 5008 7512 10016
```

If you run these applications until they finish, you should observe no error messages. The file locking ensures that only the writer process or the reader process can access temp’s 16-byte record. The other process is denied while these bytes are locked. As a result, there can be no corruption to this record’s values.
To prove to yourself that the file locking is actually working, comment out the following four lines from the previous listing, recompile `ChannelDemo.java`, and re-run this application as a writer and as a reader:

```java
FileLock lock = fc.lock(0, RECLEN, true);
FileLock lock = fc.lock(0, RECLEN, false);
lock.release();
lock.release();
```

At some point during the execution, you should observe output similar to that shown below:

```
acquiring shared lock
Reading: 803 1606 2412 3216
error
```

The output is invalid—it should be 803 1606 2409 3212 but it isn’t because the reader and writer were able to access the record at the same time.

**Note**  The more `ChannelDemo` reader processes that run, the slower a `ChannelDemo` writer process will run. Eventually, the `ChannelDemo` writer process will block during a lock acquisition attempt and not unblock because there will always be a shared lock in use.

### Mapping Files into Memory

`FileChannel` declares a `map()` method that lets you create a virtual memory mapping between a region of an open file and a `MappedByteBuffer` instance that wraps itself around this region. This mapping mechanism offers an efficient way to access a file because no time-consuming system calls are needed to perform I/O.

**Note**  `Virtual memory` is a kind of memory in which virtual addresses (also known as artificial addresses) replace physical (RAM memory) addresses. Check out Wikipedia’s “Virtual memory” topic ([http://en.wikipedia.org/wiki/Virtual_memory](http://en.wikipedia.org/wiki/Virtual_memory)) to learn more about virtual memory.
The map() method has the following signature:

`MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

The mode parameter defines the mapping mode and receives one of the following constants defined by the `FileChannel.MapMode` enumerated type:

- `READ_ONLY`: Any attempt to modify the buffer will cause `ReadOnlyBufferException` to be thrown.
- `READ_WRITE`: Changes made to the resulting buffer will eventually be propagated to the file; they may or may not be made visible to other programs that have mapped the same file.
- `PRIVATE`: Changes made to the resulting buffer will not be propagated to the file, and they will not be visible to other programs that have mapped the same file. Instead, changes will cause private copies of the modified portions of the buffer to be created. These changes are lost when the buffer is garbage collected.

The specified mapping mode is constrained by the invoking `FileChannel` object's access permissions. For example, if the file channel was opened as a read-only channel, and if you request `READ_WRITE` mode, `map()` will throw `NonWritableChannelException` because it cannot write to the file channel. Similarly, `NonReadableChannelException` is thrown when the channel was opened as write-only and you request `READ_ONLY` mode. (You can request `READ_ONLY` for a file channel opened as a read-write channel.)

**Tip** Invoke `MappedByteBuffer`'s `isReadOnly()` method to determine whether or not you can modify the mapped file.

The position and size parameters define the start and extent of the mapped region. Arguments passed to these parameters must be nonnegative. Furthermore, the argument passed to `size` must not exceed `Integer.MAX_VALUE`.

The specified range shouldn't exceed the file's size because the file will be made larger to accommodate the range. For example, if you pass `Integer.MAX_VALUE` to `size`, the file will grow to more than 2GB. Also, for a read-only mapping, `map()` will probably throw `IOException`.

The returned `MappedByteBuffer` object behaves like a memory-mapped buffer, but its contents are stored in a file. When you invoke `get()` on this object, the current contents of the file are obtained, even when these contents have been modified by an external program. Similarly, when you have write permission, invoking `put()` updates the file and changes are available to external programs.

**Note** Because mapped byte buffers are direct byte buffers, the memory space assigned to them exists outside of the virtual machine's heap.

Consider the following example:

```java
MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, 50, 100);
```
This example maps a subrange, from location 50 through location 149, of the file described by fileChannel. In contrast, the following example maps the entire file:

```
MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());
```

There is no `unmap()` method. Once a mapping is established, it remains until the `MappedByteBuffer` object is garbage collected (or the application exits, whichever happens first). Because a mapped byte buffer isn't connected to the file channel by which it was created, the mapping isn't destroyed when the file channel is closed.

`MappedByteBuffer` inherits methods from its `ByteBuffer` superclass. It also declares the following methods:

- `MappedByteBuffer load()`: Attempts to load all of the mapped file content into memory. This results in much faster access for large files because the virtual memory manager doesn't have to load portions of the file into memory as those portions are requested (by reading from/writing to their locations) while traversing the mapped buffer. Although `load()` makes a best effort, it may not succeed because external programs may cause the virtual memory manager to remove portions of the loaded file content to make room for their requests to load content into physical memory. Also, `load()` can be expensive time-wise because it can cause the virtual memory manager to perform many I/O operations; it may take time for this method to complete.

- `boolean isLoaded()`: Returns true when all of the mapped file content has been loaded into memory; otherwise, returns false. If this method returns true, you can probably access all of the content with few or no I/O operations. If this method returns false, it's still possible that buffer access will be fast and that the mapped content will be entirely resident in memory. Think of `isLoaded()` as hinting at the mapped byte buffer's status.

- `MappedByteBuffer force()`: Causes changes made to the mapped byte buffer to be written out to permanent storage. When working with mapped byte buffers, you should invoke this method instead of the file channel's `force()` method because the channel might be unaware of various changes made through the mapped byte buffer. Calling this method has no effect for `READ_ONLY` and `PRIVATE` mappings.

Listing 13-10 presents an application that demonstrates file mapping.

```
Listing 13-10. Demonstrating File Mapping

import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
```

www.it-ebooks.info
public class ChannelDemo
{
    public static void main(String[] args) throws IOException
    {
        if (args.length != 1)
        {
            System.out.println("usage: java ChannelDemo filespec");
            return;
        }
        RandomAccessFile raf = new RandomAccessFile(args[0], "rw");
        FileChannel fc = raf.getChannel();
        long size = fc.size();
        System.out.println("Size: " + size);
        MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE, 0, size);
        while (mbb.remaining() > 0)
        {
            System.out.print((char) mbb.get());
        }
        System.out.println();
        System.out.println();
        for (int i = 0; i < mbb.limit()/2; i++)
        {
            byte b1 = mbb.get(i);
            byte b2 = mbb.get(mbb.limit()-i-1);
            mbb.put(i, b2);
            mbb.put(mbb.limit()-i-1, b1);
        }
        mbb.flip();
        while (mbb.remaining() > 0)
        {
            System.out.print((char) mbb.get());
        }
        fc.close();
    }
}

After verifying that you’ve specified a single command-line argument, which should identify an existing file, main() creates a RandomAccessFile object for accessing this file in read/write mode. It then obtains a file channel for communicating with this file.

After using the file channel to obtain the file size, which is subsequently output, main() uses the file channel to invoke map() to obtain a read/write mapping of the entire file. It subsequently outputs the contents of the returned mapped byte buffer.

Later on, main() enters a for loop whose purpose is to reverse the file’s contents. In each of the iterations, two bytes that mirror each other are obtained and then swapped. After leaving this loop, main() flips the buffer for draining and outputs its reversed contents.

Compile Listing 13-10 (javac ChannelDemo.java) and, assuming the existence of a poem.txt file, execute the following command line to reverse this file’s contents:

java ChannelDemo poem.txt
CHAPTER 13: Migrating to New I/O

You should observe output similar to the following:

Size: 67
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you!

!uoy era os dnA
,teews si ragu$,
eulb era steloiV,
,der era sesoR

The blank lines in the reversed text result from reversing the carriage return (13)/line feed (10) sequences on Windows platforms. Also, the contents of poem.txt should be reversed.

Transferring Bytes Among Channels

To optimize the common practice of performing bulk transfers, two methods have been added to FileChannel that avoid the need for intermediate buffers:

- long transferFrom(ReadableByteChannel src, long position, long count):
  Transfers bytes into this channel's file from the given readable byte channel. Parameter src identifies the source channel, position identifies the nonnegative start position in the file where the transfer is to start, and count identifies the nonnegative maximum number of bytes that are to be transferred. This method returns the number of bytes (possibly 0) that were actually transferred. It throws IllegalArgumentException when a precondition on a parameter (such as position being nonnegative) doesn't hold; NonReadableChannelException when the source channel wasn't opened for reading; NonWritableChannelException when this channel wasn't opened for writing; ClosedChannelException when this channel or the source channel is closed; ClosedByInterruptException when another thread interrupts the current thread while the transfer is in progress, thereby closing both channels and setting the current thread's interrupt status; and IOException when some other I/O error occurs.

- long transferTo(long position, long count, WritableByteChannel target):
  Transfers bytes from this channel's file to the given writable byte channel. Parameter position identifies the nonnegative start position in the file where the transfer is to start, count identifies the nonnegative maximum number of bytes that are to be transferred, and target identifies the target channel. This method returns the number of bytes (possibly 0) that were actually transferred. It throws IllegalArgumentException when a precondition on a parameter doesn't hold; NonReadableChannelException when this channel wasn't opened for reading; NonWritableChannelException when this channel wasn't opened for writing; and IOException when some other I/O error occurs.
writing; ClosedChannelException when this channel or the target channel is closed; ClosedByInterruptException when another thread interrupts the current thread while the transfer is in progress, thereby closing both channels and setting the current thread's interrupt status; and IOException when some other I/O error occurs.

If you're using transferTo() with a file channel as the transfer source, the transfer stops at the end of the file when position plus count exceeds the file's size. Similarly, transferFrom() stops when src is a file channel and its end of file is reached.

Listing 13-11 presents an application that demonstrates channel transfer.

Listing 13-11. Demonstrating Channel Transfer

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.channels.Channels;
import java.nio.channels.FileChannel;
import java.nio.channels.WritableByteChannel;

public class ChannelDemo {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java ChannelDemo filespec");
            return;
        }

        FileInputStream fis = null;
        try {
            fis = new FileInputStream(args[0]);
            FileChannel inChannel = fis.getChannel();
            WritableByteChannel outChannel = Channels.newChannel(System.out);
            inChannel.transferTo(0, inChannel.size(), outChannel);
        }
        catch (IOException ioe) {
            System.out.println("I/O error: " + ioe.getMessage());
        }
        finally {
            if (fis != null)
                try {
                    fis.close();
                }
        }
    }
}
```
Listing 13-11’s main() method verifies that a single command-line argument has been specified. This argument identifies a file whose contents are to be copied to the standard output stream.

Next, main() creates a file input stream to the file identified by the command-line argument and a file channel for reading from this file.

Finally, an output channel for sending bytes to the standard output stream is obtained and the input file channel’s transferTo() method is called to transfer the file content to standard output.

Compile Listing 13-11 (javac ChannelDemo.java), and execute the following command line to make a copy of ChannelDemo.java:

```
java ChannelDemo ChannelDemo.java >ChannelDemo.bak
```

You should observe a ChannelDemo.bak file with size and contents identical to ChannelDemo.java.

**Socket Channels**

I previously mentioned that Socket declares a SocketChannel getChannel() method for returning a socket channel instance, which describes an open connection to a socket. Unlike sockets, socket channels are selectable and can function in nonblocking mode. These capabilities (discussed later in this chapter) enhance the scalability and flexibility of large applications (such as web servers).

---

**Note** Although you might not care about scalability and flexibility when developing an Android app, suppose that your app needs to communicate with a custom web server that you must also create. When many instances of your app need to simultaneously communicate with the server, you’ll appreciate being able to leverage the selectable and nonblocking mode capabilities of socket channels to ensure that the server is responsive to each app instance. After all, why risk angry customers?

Socket channels are described by the abstract java.nio.channels.ServerSocketChannel, java.nio.channels.SocketChannel, and java.nio.channels.DatagramChannel classes. Each class ultimately extends java.nio.channels.SelectableChannel and implements InterruptibleChannel, making ServerSocketChannel, SocketChannel, and DatagramChannel instances selectable and interruptible. Because SocketChannel and DatagramChannel implement the ByteChannel, GatheringByteChannel, and ScatteringByteChannel interfaces, you can write to, read from, and perform scattering I/O on their underlying sockets.
Note Unlike buffers, which are not thread-safe, server socket channels, socket channels, and datagram channels are thread-safe.

Each ServerSocketChannel, SocketChannel, and DatagramChannel instance creates a peer socket object from the java.net.ServerSocket, Socket, or java.net.DatagramSocket class; each class has been retrofitted to work with channels. You can obtain the peer socket object by invoking ServerSocketChannel’s, SocketChannel’s, or DatagramChannel’s socket() method.

Note When invoked on the socket instance returned from socket(), getChannel() returns the associated socket channel. However, when invoked on a socket obtained by instantiating ServerSocket, Socket, or DatagramSocket, getChannel() returns null.

Understanding Nonblocking Mode

The blocking nature of sockets created from Java’s socket classes is a serious limitation to a network-oriented Java application’s scalability. For example, the ServerSocket class’s Socket accept() method blocks until an incoming connection arrives, at which point it creates and returns a Socket instance that lets the server communicate with the client. If this method didn’t block, scalability would improve because the server could be accomplishing other useful work instead of having to wait.

The abstract SelectableChannel class is a common ancestor of the ServerSocketChannel, SocketChannel, and DatagramChannel classes. In addition to letting the socket channel work in a selector context (I discuss selectors later in this chapter), SelectableChannel lets socket channels choose to block or operate in nonblocking mode.

Note SelectableChannel merges functionality related to selectors with nonblocking mode because nonblocking mode is most useful in conjunction with selector-based multiplexing.

SelectableChannel offers the following methods for enabling blocking or nonblocking, determining whether the channel is blocking or nonblocking, and obtaining the blocking lock:

SelectableChannel configureBlocking(boolean block): Specifies the calling selectable channel’s blocking status. Pass true to make the channel blocking and false to make the channel nonblocking. The method returns the selectable channel or throws an exception: ClosedChannelException when the channel is closed, java.net.channels.IllegalBlockingModeException when block is true and the channel has been registered with one or more selectors, and IOException when an I/O error occurs.
- **boolean isBlocking()**: This method returns true when the calling selectable channel is blocking; otherwise, false returns. Newly created channels default to blocking.

- **Object blockingLock()**: Returns the object on which `configureBlocking()` synchronizes. The returned object is useful in the implementation of adaptors that require the current blocking mode value to not change for a short period of time.

It's trivial to set or reset a selectable channel's blocking/nonblocking status. To enable nonblocking, pass false to an invocation of `configureBlocking()`, which the following example demonstrates:

```java
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking(false); // enable nonblocking mode
```

Although nonblocking sockets are commonly used in server-oriented applications, they are also beneficial on the client side. For example, a GUI application can leverage nonblocking sockets to keep the user interface responsive while communicating simultaneously with several server applications.

The `blockingLock()` method lets you prevent other threads from changing a socket channel's blocking/nonblocking status. This method returns the object that a channel implementation uses for synchronizing when changing this status. Only the thread that holds the lock on this object can change the status, and the lock is often obtained by using Java's `synchronized` keyword. Consider the following example:

```java
ServerSocketChannel ssc = ServerSocketChannel.open();
SocketChannel sc = null;
Object lock = ssc.blockingLock();

// Thread might block when obtaining the lock associated with
// the lock object.

synchronized(lock)
{
    // Current thread owns the lock. No other thread can
    // change blocking mode.

    // Obtaining server socket channel's current blocking mode.
    boolean blocking = ssc.isBlocking();

    // Set server socket channel to nonblocking.
    ssc.configureBlocking(false);

    // Obtain next connection, which is null when there is no
    // connection.
    sc = ssc.accept();

    // Restore previous blocking mode.
    ssc.configureBlocking(blocking);
}
```
The lock is released and some other thread may modify the
server socket channel's blocking mode.

if (sc != null)
    communicateWithSocket(sc);

**Exploring Server Socket Channels**

ServerSocketChannel is the simplest of the three socket channel classes. This class includes the following methods:

- **static ServerSocketChannel open():** Attempts to open a server-socket channel, which is initially unbound; it must be bound to a specific address via one of its peer socket’s bind() methods before connections can be accepted. If the channel cannot be opened, IOException is thrown.

- **ServerSocket socket():** Returns the peer ServerSocket instance associated with this server socket channel.

- **SocketChannel accept():** Accepts the connection made to this channel’s socket. If this channel is nonblocking, accept() immediately returns null when there are no pending connections or returns a socket channel that represents the connection. Otherwise, when the channel is blocking, accept() blocks indefinitely until a new connection is available or an I/O error occurs. The socket channel returned by this method is blocking regardless of whether the server socket channel is blocking or nonblocking. This method throws ClosedChannelException when the server socket channel is closed, AsynchronousCloseException when another thread closes this server socket channel while the accept operation is in progress, java.nio.channels.NotYetBoundException when the server socket channel hasn’t been bound, or IOException when an I/O error occurs.

A server socket channel behaves as a server in the TCP/IP stream protocol. You use server socket channels to listen for incoming connections with clients.

You create a new server socket channel by invoking the static open() factory method. If all goes well, open() returns a ServerSocketChannel instance associated with an unbound peer ServerSocket object. You can obtain this object by invoking socket(), and then invoke ServerSocket’s bind() method to bind the server socket (and ultimately the server socket channel) to a specific address.

You can then invoke ServerSocketChannel’s accept() method to accept an incoming connection. Depending on whether or not you have configured the server socket channel to be nonblocking, this method either returns immediately with null or a socket channel to an incoming connection, or blocks until there is an incoming connection.

**Note** Alternatively, you can invoke accept() on the peer ServerSocket object that socket() returns. However, this accept() method will always block.
Listing 13-12 presents a ChannelServer application that demonstrates ServerSocketChannel.

**Listing 13-12. Demonstrating ServerSocketChannel**

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;

public class ChannelServer {
    public static void main(String[] args) throws IOException {
        System.out.println("Starting server...");
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.socket().bind(new InetSocketAddress(9999));
        ssc.configureBlocking(false);
        String msg = "Local address: " + ssc.socket().getLocalSocketAddress();
        ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
        while (true) {
            System.out.print(".");
            SocketChannel sc = ssc.accept();
            if (sc != null) {
                System.out.println();
                System.out.println("Received connection from " +
                    sc.socket().getRemoteSocketAddress());
                buffer.rewind();
                sc.write(buffer);
                sc.close();
            } else {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException ie) {
                    assert false; // shouldn't happen
                }
            }
        }
    }
}
```

Listing 13-12's `main()` method first outputs a startup message and then obtains a server socket channel. Continuing, it accesses the ServerSocket peer object and uses this object to bind the socket/channel to port 9999.
Next, main() configures the server socket channel to be nonblocking and creates a byte buffer based on a message that identifies the server socket channel's local socket address.

main() now enters a while loop that repeatedly prints a single period character to demonstrate the channel's nonblocking status and checks for an incoming connection. If a connection is detected, its SocketChannel instance is used to obtain the remote socket address, which is output to the standard output stream. The buffer is then rewound and its content written to the socket channel, which is then closed. However, if a connection isn't detected, main() sleeps for a fraction of a second.

Compile Listing 13-12 as follows:

```java
javac ChannelServer.java
```

Execute the following command to start the server:

```bash
java ChannelServer
```

You should observe a starting server... message followed by a growing sequence of periods across the screen from left to right. At this point, there's nothing further to observe.

### Exploring Socket Channels

SocketChannel is the most commonly used of the three socket channel classes, and it models a connection-oriented stream protocol (such as TCP/IP). This class includes the following methods:

- `static SocketChannel open()`: Attempts to open a socket channel. If the channel cannot be opened, IOException is thrown.
- `static SocketChannel open(InetSocketAddress remoteAddr)`: Attempts to open a socket channel and connect it to remoteAddr. This convenience method works as if by invoking the open() method, invoking the connect() method upon the resulting socket channel, passing it remoteAddr, and then returning that channel. This method throws AsynchronousCloseException when another thread closes this channel while the connect operation is in progress; ClosedByInterruptException when another thread interrupts the current thread while the connect operation is in progress, thereby closing the channel and setting the current thread's interrupt status; java.nio.channels.UnresolvedAddressException when the given remote address isn't fully resolved; java.nio.channels.UnsupportedAddressTypeException when the type of the given remote address isn't supported; and IOException when some other I/O error occurs.
- `Socket socket()`: Returns the peer Socket instance associated with this socket channel.
- `boolean connect(SocketAddress remoteAddr)`: Attempts to connect this socket channel's socket object to the remote address. If this channel is nonblocking, an invocation of this method initiates a nonblocking connection operation. If the connection is established immediately, as can happen with a local connection, this method returns true. Otherwise, this method returns false and the connection operation must be subsequently completed by repeatedly...
invoking the finishConnect() method until this method returns true. This method throws java.nio.channels.AlreadyConnectedException when this channel is already connected; java.nio.channels.ConnectionPendingException when a nonblocking connection operation is already in progress on this channel; ClosedChannelException when this channel is closed; AsynchronousCloseException when another thread closes this channel while the connect operation is in progress; ClosedByInterruptException when another thread interrupts the current thread while the connect operation is in progress, thereby closing the channel and setting the current thread's interrupt status; UnresolvedAddressException when the given remote address isn't fully resolved; UnsupportedAddressTypeException when the type of the given remote address isn't supported; and IOException when some other I/O error occurs.

- boolean isConnectionPending(): Returns true when a connection operation is pending completion; otherwise, returns false.

- boolean finishConnect(): Finishes the process of connecting a socket channel. This method returns true when the socket channel is fully connected; otherwise, returns false. This method throws java.nio.channels.NoConnectionPendingException when this channel isn't connected and a connection operation hasn't been initiated; ClosedChannelException when this channel is closed; AsynchronousCloseException when another thread closes this channel while the connect operation is in progress; ClosedByInterruptException when another thread interrupts the current thread while the connect operation is in progress, thereby closing the channel and setting the current thread's interrupt status; and IOException when some other I/O error occurs.

- boolean isConnected(): Returns true when this channel's socket is open and connected; otherwise, returns false.

A socket channel behaves as a client in the TCP/IP stream protocol. You use socket channels to initiate connections to listening servers.

Create a new socket channel by calling either of the open() methods. Behind the scenes a peer Socket object is created. Invoke SocketChannel's socket() method to return this peer object. Also, you can return the original socket channel by invoking getChannel() on the peer Socket object.

A socket channel obtained from the noargument open() method isn't connected. Attempting to read from or write to this socket channel results in java.nio.channels.NotYetConnectedException. To connect the socket, call the connect() method on the socket channel or on its peer socket.

After a socket channel has been connected, it remains connected until closed. To determine if a socket channel is connected, invoke SocketChannel's boolean isConnected() method.

The open() method that takes a java.net.InetAddress argument also lets you connect to another host at the specified remote address, as follows:

SocketChannel sc = SocketChannel.open(new InetSocketAddress("localhost", 9999));
This convenience method is equivalent to invoking the following code sequence:

```java
SocketChannel sc = SocketChannel.open();
sc.connect(new InetSocketAddress("localhost", 9999));
```

When connecting to a server via the peer `Socket` object or via `SocketChannel`'s `connect()`/second `open()` method on a blocking socket channel, the thread that invokes `connect()` blocks until the socket channel is connected. However, when the socket channel isn't blocking, `connect()` returns immediately, typically with false to indicate that the connection hasn’t been made (although it might return true for a local loopback connection). Because a connection must be established before you can perform I/O on the socket channel, you need to invoke `finishConnect()` repeatedly until this method returns true.

Listing 13-13 presents a `ChannelClient` application that demonstrates `SocketChannel`.

**Listing 13-13. Demonstrating `SocketChannel`**

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
public class ChannelClient {
    public static void main(String[] args) {
        try {
            SocketChannel sc = SocketChannel.open();
            sc.configureBlocking(false);
            InetSocketAddress addr = new InetSocketAddress("localhost", 9999);
            sc.connect(addr);
            while (!sc.finishConnect())
                System.out.println("waiting to finish connection");

            ByteBuffer buffer = ByteBuffer.allocate(200);
            while (sc.read(buffer) >= 0) {
                buffer.flip();
                while (buffer.hasRemaining())
                    System.out.print((char) buffer.get());
                buffer.clear();
            }
            sc.close();
        }
    }
}
```
Listing 13-13’s `main()` method first obtains a socket channel and configures it to be nonblocking. It then creates an address to the previous channel server application and initiates a connection to this address. Because of the nonblocking status, it’s necessary to invoke `finishConnect()` repeatedly until this method returns true, which indicates a connection to the remote server application.

`main()` subsequently creates a byte buffer and enters a loop that repeatedly reads content into this buffer and outputs this content to the standard output stream. The channel is then closed.

Compile Listing 13-13 via the following command:

```
javac ChannelClient.java
```

Assuming that the channel server is running, execute the following command to start the client:

```
java ChannelClient
```

You should observe a message similar to the following on the channel server output stream:

```
Received connection from /127.0.0.1:51177
```

You should also observe the following message on the channel client output stream:

```
Local address: /0:0:0:0:0:0:0:0:9999
```

### Exploring Datagram Channels

`DatagramChannel` models a connectionless packet-oriented protocol (such as UDP/IP). This class includes the following methods:

- `static DatagramChannel open()`: Attempts to open a datagram channel. If the channel cannot be opened, `IOException` is thrown.
- `DatagramSocket socket()`: Returns the peer `DatagramSocket` instance associated with this datagram channel.
- `DatagramChannel connect(SocketAddress remoteAddr)`: Attempts to connect this datagram channel’s socket object to the remote address. The channel’s socket is configured so that it only receives datagrams from and sends datagrams to the given address. Once connected, datagrams cannot be received from or sent to any other address. A datagram socket remains connected until explicitly disconnected or closed. This method returns the datagram channel upon success. It throws `ClosedChannelException` when the datagram channel is closed; `AsynchronousCloseException` when another thread closes this channel.
while the connect operation is in progress; ClosedByInterruptException when another thread interrupts the current thread while the connect operation is in progress, thereby closing the channel and setting the current thread’s interrupt status; and IOException when some other I/O error occurs.

- boolean isConnected(): Returns true when this channel’s socket is open and connected; otherwise, returns false.

- DatagramChannel disconnect(): Disconnects this channel’s socket. This method may be invoked at any time and has no effect on read or write operations that are already in progress. When the socket isn’t connected or when the channel is closed, invoking this method has no effect.

- SocketAddress receive(ByteBuffer buffer): Receives a datagram via this channel. If a datagram is immediately available or if this channel is blocking and a datagram becomes available, the datagram is copied into the given byte buffer and its source address is returned. If this channel is nonblocking and a datagram isn’t immediately available, this method immediately returns null. The datagram is transferred into the given byte buffer starting at its current position, as if by a regular read operation. If there are fewer bytes remaining in the buffer than are required to hold the datagram, the remainder of the datagram is silently discarded. This method returns the datagram’s source address or null when the channel isn’t blocking and no datagram is available. It throws ClosedChannelException when the datagram channel is closed; AsynchronousCloseException when another thread closes this channel while the read operation is in progress; ClosedByInterruptException when another thread interrupts the current thread while the read operation is in progress, thereby closing the channel and setting the current thread’s interrupt status; and IOException when some other I/O error occurs.

- int send(ByteBuffer buffer, SocketAddress destAddr): Sends a datagram via this channel. If this channel is nonblocking and there is sufficient room in the underlying output buffer, or if this channel is blocking and sufficient room becomes available, the remaining bytes in the given buffer are transmitted as a single datagram to the given destination address. The datagram is transferred from the byte buffer as if by a regular write operation. This method returns the number of bytes sent, which will be the number of bytes that were remaining in the source buffer when this method was invoked or, when this channel is nonblocking, may be zero if there was insufficient room for the datagram in the underlying output buffer. It throws ClosedChannelException when the datagram channel is closed; AsynchronousCloseException when another thread closes this channel while the write operation is in progress; ClosedByInterruptException when another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread’s interrupt status; and IOException when some other I/O error occurs.

Additionally, there are several read() and write() methods that you might like to use. Unlike send() and receive(), which don’t require the datagram channel to be connected, the read() and write() methods require a connection.
As with ServerSocketChannel and SocketChannel, you obtain a DatagramChannel instance by invoking the static open() method. The new datagram channel is associated with a peer DatagramSocket object, which you can obtain by invoking DatagramChannel's socket() method.

A datagram channel can behave as both a client (the sender) and a server (the listener). To act as a listener, the datagram channel must be bound to a port and an optional address. Accomplish this task by obtaining the DatagramSocket object and invoking bind() on this object, as follows:

```java
DatagramChannel dc = DatagramChannel.open();
DatagramSocket ds = dc.socket();
ds.bind(new InetSocketAddress(9999)); // bind to port 9999
```

The receive() method copies the incoming datagram’s data payload into the byte buffer argument and returns a socket address identifying the datagram’s source address. If the channel is blocking, receive() sleeps until the packet arrives or some event results in a thrown exception. If the channel is nonblocking, receive() returns null when a datagram isn’t available. If the data payload is larger than will fit in the buffer, excess bytes are quietly removed.

The send() method sends the given byte buffer’s content, starting from the current position and ranging to the buffer’s limit, to the destination address/port number specified by the socket address argument. If the datagram channel is blocking, send() sleeps until the datagram is queued for sending or some event results in a thrown exception. If the channel isn’t blocking, this method returns with one of two values: the entire length of the buffer content that was sent or 0 indicating that the buffer content wasn’t sent—nothing is sent when there isn’t room to store the entire datagram before transmission.

**Note**  Datagram protocols aren’t reliable. For one thing, they don’t guarantee delivery. As a result, a nonzero return value from send() doesn’t mean that the datagram reached its destination. Also, the underlying network might fragment the datagram into multiple smaller packets. When a datagram is fragmented, it’s more probable for one or more of these packets not to arrive at the destination. Because the receiver cannot reassemble all of the packets, the entire datagram is discarded. For this reason, data payloads should be restricted to several hundred bytes maximum.

An example of where you might require a datagram channel is a stock ticker that offers the latest stock prices for a given company. A client would submit a company’s stock symbol (such as MSFT for Microsoft) as a datagram payload and receive a datagram in response whose payload provides the requested stock prices. Because the latest information is desired, the client would re-request the stock prices when a response datagram doesn’t arrive.

Listing 13-14 presents a ChannelServer application that leverages DatagramChannel to implement the server portion of the stock ticker.
Listing 13-14. Using DatagramChannel to Implement a Stock Ticker Server

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class ChannelServer
{
    final static int PORT = 9999;

    public static void main(String[] args) throws IOException
    {
        System.out.println("server starting and listening on port " +
                      PORT + " for incoming requests...");
        DatagramChannel dcServer = DatagramChannel.open();
        dcServer.socket().bind(new InetSocketAddress(PORT));
        ByteBuffer symbol = ByteBuffer.allocate(4);
        ByteBuffer payload = ByteBuffer.allocate(16);
        while (true)
        {
            payload.clear();
            symbol.clear();
            SocketAddress sa = dcServer.receive(symbol);
            if (sa == null)
                return;
            System.out.println("Received request from " + sa);
            String stockSymbol = new String(symbol.array(), 0, 4);
            System.out.println("Symbol: " + stockSymbol);
            if (stockSymbol.toUpperCase().equals("MSFT"))
            {
                payload.putFloat(0, 37.40f); // open share price
                payload.putFloat(4, 37.22f); // low share price
                payload.putFloat(8, 37.48f); // high share price
                payload.putFloat(12, 37.41f); // close share price
            }
            else
            {
                payload.putFloat(0, 0.0f);
                payload.putFloat(4, 0.0f);
                payload.putFloat(8, 0.0f);
                payload.putFloat(12, 0.0f);
            }
            dcServer.send(payload, sa);
        }
    }
}
```

Listing 13-14's main() method first creates a datagram channel and binds it to port 9999. It then creates two byte buffers to hold a 4-byte stock symbol and a 16-byte response, which is organized into four 4-byte floating-point values representing open share price, low share price, high share price, and close share price.

```
Note For convenience, I'm representing currency amounts as floating-point values. This is not a good idea in practice and java.math.BigDecimal should be used instead. Also, you wouldn't embed stock prices in the source code but would dynamically obtain them from some kind of external server or database.
```

main() now enters an infinite loop that clears both byte buffers in preparation for receiving new information from a client and receives the next stock symbol. The subsequent if statement that tests sa for null isn’t necessary for this application but is present in case you want to configure the channel for nonblocking mode.

After outputting a message identifying the request, main() checks the stock symbol to see if it equals MSFT. If so, the payload byte buffer is configured to store four stock prices for Microsoft stock; otherwise, the payload byte buffer is configured to store four 0 prices (to indicate unknown stock symbol).

Finally, main() sends the payload datagram payload to the receiver and continues to loop.

Compile Listing 13-14 via the following command:

```
javac ChannelServer.java
```

Run the application as follows:

```
java ChannelServer
```

You should observe the following message:

```
server starting and listening on port 9999 for incoming requests...
```

Listing 13-15 presents a ChannelClient application that leverages DatagramChannel to implement the client portion of the stock ticker.

```
Listing 13-15. Using DatagramChannel to Implement a Stock Ticker Client

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class ChannelClient
{
    final static int PORT = 9999;
```
public static void main(String[] args) throws IOException
{
    if (args.length != 1)
    {
        System.err.println("usage: java ChannelClient stocksymbol");
        return;
    }

    DatagramChannel dcClient = DatagramChannel.open();

    ByteBuffer symbol = ByteBuffer.wrap(args[0].getBytes());
    ByteBuffer response = ByteBuffer.allocate(16);

    InetSocketAddress sa = new InetSocketAddress("localhost", PORT);
    dcClient.send(symbol, sa);
    System.out.println("Receiving datagram from " +
                        dcClient.receive(response));
    System.out.println("Open price: " + response.getFloat(0));
    System.out.println("Low price: " + response.getFloat(4));
    System.out.println("High price: " + response.getFloat(8));
    System.out.println("Close price: " + response.getFloat(12));
}

Listing 13-15’s main() method first verifies that a single command-line argument has been specified. This argument identifies a stock symbol. It then creates a datagram channel and a pair of byte buffers: symbol stores the specified symbol and response stores the response from the server.

Next, main() creates a socket address for communicating with and sends the symbol buffer to the server. It then receives a response datagram from the server, storing its payload in the response buffer.

Finally, main() accesses the response buffer, using getFloat() to convert each set of 4 bytes to a floating-point value, which is subsequently output.

Compile Listing 13-15 via the following command:

javac ChannelClient.java

Run this application as follows:

java ChannelClient msft

Assuming that the server is still running, you should observe the following messages in the server window:

---

Received request from /127.0.0.1:64837
Symbol: msft
---
Pipes

The java.nio.channels package includes a Pipe class. Pipe describes a pair of channels that implement a unidirectional pipe, which is a conduit for passing data in one direction between two entities, such as two file channels or two socket channels. Pipe is analogous to the java.io.PipedInputStream and java.io.PipedOutputStream classes—see Chapter 11.

This class declares nested SourceChannel and SinkChannel classes that serve as readable and writable byte channels, respectively. Pipe also declares the following methods:

- static Pipe open(): This class method opens a new pipe, throwing IOException when an I/O error occurs.
- SourceChannel source(): This method returns the pipe’s source channel.
- SinkChannel sink(): This method returns the pipe’s sink channel.

Pipes can be used to pass data within the same virtual machine; you cannot use them to pass data between the virtual machine and an external program. Pipes are ideal in producer/consumer scenarios because of encapsulation: you can use the same code to write data to files, sockets, or pipes depending upon the kind of channel presented to the pipe.

Listing 13-16 presents a producer/consumer application that uses a pipe to achieve communication between two threads.

Listing 13-16. Producing and Consuming Bytes via a Pipe

```java
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.Pipe;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;

public class ChannelDemo {
    final static int BUFSIZE = 10;
    final static int LIMIT = 3;

    public static void main(String[] args) throws IOException {
        final Pipe pipe = Pipe.open();
```
Runnable senderTask = new Runnable()
{
    @Override
    public void run()
    {
        WritableByteChannel src = pipe.sink();
        ByteBuffer buffer = ByteBuffer.allocate(BUFSIZE);
        for (int i = 0; i < LIMIT; i++)
        {
            buffer.clear();
            for (int j = 0; j < BUFSIZE; j++)
                buffer.put((byte) (Math.random() * 256));
            buffer.flip();
            try
            {
                while (src.write(buffer) > 0);
            }
            catch (IOException ioe)
            {
                System.err.println(ioe.getMessage());
            }
        }
    try
    {
        src.close();
    }
    catch (IOException ioe)
    {
    }
}
}

Runnable receiverTask = new Runnable()
{
    @Override
    public void run()
    {
        ReadableByteChannel dst = pipe.source();
        ByteBuffer buffer = ByteBuffer.allocate(BUFSIZE);
        try
        {
            while (dst.read(buffer) >= 0)
            {
                buffer.flip();
                while (buffer.remaining() > 0)
                    System.out.println(buffer.get()&255);
                buffer.clear();
            }
        }
    }
}
catch (IOException ioe)
{
    System.err.println(ioe.getMessage());
}
}
Thread sender = new Thread(senderTask);
Thread receiver = new Thread(receiverTask);
sender.start();
receiver.start();
}

Listing 13-16's main() method first obtains a pipe and then creates sender and receiver tasks that serve as producer and consumer. main() then creates sender and receiver threads and starts them.

The sender task’s run() method first obtains a writable byte channel from the pipe by invoking Pipe’s sink() method. It then allocates a byte buffer for storing content to be written.

run() continues by entering a pair of for loops for sending byte-oriented data to the writable byte channel. Each of the outer for loop iterations clears the buffer in preparation for filling by the inner for loop. The buffer is then flipped in preparation for draining, which is accomplished by passing the buffer to the writable byte channel’s write() method. Because a single method call might not drain the entire buffer, write() is invoked in a loop until it returns 0, which means that there is no more content to write. The channel is then closed so that the receiver task doesn’t block when reading from the channel because it expects to receive more data.

The receiver task’s run() method first obtains a readable byte channel from the pipe by invoking Pipe’s source() method. It then allocates a buffer for storing read content.

Continuing, run() enters a while loop that continually reads from the channel until the read() method returns -1, which indicates that the channel has reached the end of the stream. This method wouldn’t reach the end of the stream if the sender’s run() method hadn’t closed the channel.

At this point, the buffer is flipped to prepare it for draining. It’s then drained by printing its byte values to the standard output stream. Each byte is bitwise ANDed with 255 to prevent a negative value from being output. Basically, get() returns an 8-bit integer value that’s converted to a 32-bit integer during the System.out.println() method call. This conversion applies sign extension, which means that some byte values become negative 32-bit integers. By bitwise ANDing the byte value with 255, the conversion ensures that no byte value is turned into a negative 32-bit integer.

Finally, the buffer is cleared in preparation for filling and the loop continues.
CHAPTER 13: Migrating to New I/O

Compile Listing 13-16 (javac ChannelDemo.java), and run the application (java ChannelDemo).
You should observe a sequence of 30 random integers similar to the following:

245
56
137
166
52
183
252
166
246
124
163
11
159
68
203
118
157
70
54
148
186
17
12
203
75
223
224
175
205
47

Working with Selectors

I/O is either block-oriented (such as file I/O) or stream-oriented (such as network I/O). Streams are
often slower than block devices (such as fixed disks) and read/write operations often cause the
calling thread to block until input is available or output has been fully written. To compensate,
modern operating systems let streams operate in nonblocking mode, which makes it possible for a
thread to read or write data without blocking. The operation fully succeeds, or it indicates that the
read/write isn’t possible at that time. Either way, the thread is able to perform other useful work
instead of waiting.

Nonblocking mode doesn’t let an application determine if it can perform an operation without
actually performing the operation. For example, when a nonblocking read operation succeeds, the
application learns that the read operation is possible but also has read some data that must be
managed. This duality prevents you from separating code that checks for stream readiness from the
data-processing code without making your code significantly complicated.
Nonblocking mode serves as a foundation for performing *readiness selection*, which offloads to the operating system the work involved in checking for I/O stream readiness to perform write, read, and other operations. The operating system is instructed to observe a group of streams and return some indication of which streams are ready to perform a specific operation (such as read) or operations (such as accept and read). This capability lets a thread *multiplex* a potentially huge number of active streams by using the readiness information provided by the operating system. In this way, network servers can handle large numbers of network connections; they are vastly scalable.

**Note**  Modern operating systems make readiness selection available to applications by providing system calls such as the POSIX `select()` call.

Selectors let you achieve readiness selection in a Java context. In this section, I first introduce you to selector fundamentals and then provide a demonstration.

**Selector Fundamentals**

A *selector* is an object created from a subclass of the abstract `java.nio.channels.Selector` class. It maintains a set of channels, which it examines to determine which of them are ready for reading, writing, completing a connection sequence, accepting another connection, or some combination of these tasks. The actual work is delegated to the operating system via a POSIX `select()` or similar system call.

**Note**  The ability to check a channel without having to wait when something isn’t ready (such as bytes are not available for reading) and without also having to perform the operation while checking is the key to scalability. A single thread can manage a huge number of channels, which reduces code complexity and potential threading issues.

Selectors are used with *selectable channels*, which are objects whose classes ultimately inherit from the abstract `SelectableChannel` class, which describes a channel that can be multiplexed by a selector. Socket channels, server socket channels, datagram channels, and pipe source/sink channels are selectable channels because `SocketChannel`, `ServerChannel`, `DatagramChannel`, `Pipe.SinkChannel`, and `Pipe.SourceChannel` are derived from `SelectableChannel`. In contrast, file channels are not selectable channels because `FileChannel` doesn’t include `SelectableChannel` in its ancestry.

One or more previously created selectable channels are registered with a selector. Each registration returns a *key* (described by a concrete instance of the abstract `java.nio.channels.SelectionKey` class) that’s a token signifying the relationship between one channel and the selector. This key keeps track of two sets of operations: interest set and ready set. The *interest set* identifies the operation categories that will be tested for readiness the next time one of the selector’s selection methods is invoked. The *ready set* identifies the operation categories for which the key’s channel has been found to be ready by the key’s selector. When a selection method is invoked, the selector’s associated keys are updated by checking all channels registered with that selector. The application
can then obtain a set of keys whose channels were found ready, and iterate over these keys to service each channel that has become ready since a previous select method call.

**Note** A selectable channel can be registered with more than one selector. It has no knowledge of the selectors to which it's currently registered.

To work with selectors, you first need to create one. You can accomplish this task by invoking `Selector`'s `Selector open()` class method. This method returns a `Selector` instance on success or throws `IOException` on failure. The following code fragment demonstrates this task:

```java
Selector selector = Selector.open();
```

You can create your selectable channels before or after creating the selector. However, you must ensure that each channel is in nonblocking mode before registering the channel with the selector. You register a selectable channel with a selector by invoking either of the following `SelectableChannel` registration methods:

- `SelectionKey register(Selector sel, int ops)`
- `SelectionKey register(Selector sel, int ops, Object att)`

Each method requires that you pass a previously created selector to `sel` and a bitwise ORed combination of the following `SelectionKey` int-based constants to `ops`, which signifies the interest set:

- `OP_ACCEPT`: Operation-set bit for socket-accept operations.
- `OP_CONNECT`: Operation-set bit for socket-connect operations.
- `OP_READ`: Operation-set bit for read operations.
- `OP_WRITE`: Operation-set bit for write operations.

The second method also lets you pass an arbitrary `java.lang.Object`/subclass instance (or null) to `att`. The non-null object is known as an *attachment*, and it is a convenient way of recognizing a given channel or attaching additional information to the channel. It's stored in the `SelectionKey` instance returned from this method.

Upon success, each method returns a `SelectionKey` instance that relates the selectable channel with the selector. Upon failure, an exception is thrown. For example, `ClosedChannelException` is thrown when the channel is closed and `IllegalBlockingModeException` is thrown when the channel hasn't been set to nonblocking mode.

The following code fragment extends the previous code fragment by configuring a previously created channel to nonblocking mode and registering the channel with the selector, whose selection methods are to test the channel for accept, read, and write readiness:

```java
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_ACCEPT | SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```
At this point, the application typically enters an infinite loop where it accomplishes the following tasks.

1. Performs a selection operation.
2. Obtains the selected keys followed by an iterator over the selected keys.
3. Iterates over these keys and perform channel operations.

A selection operation is performed by invoking one of Selector's selection methods. For example, int select() performs a blocking selection operation. It doesn't return until at least one channel is selected, this selector's wakeup() method is invoked, or the current thread is interrupted, whichever comes first.

A set of the selected keys (the ready set) is now obtained by invoking Selector's Set<SelectionKey> selectedKeys() method. Invoke the set's iterator() method to obtain an iterator over these keys.

Finally, the application iterates over the keys. For each of the iterations, a SelectionKey instance is returned. Some combination of SelectionKey's boolean isAcceptable(), boolean isConnectable(), boolean isReadable(), and boolean isWritable() methods are called to determine if the key indicates that a channel is ready to accept a connection, finished connecting, readable, or writable.

The select() method returns the number of channels that have become ready since the last time it was called. For example, if you call select() and it returns 1 because one channel has become ready, and if you call select() again and a second channel has become ready, select() will once again return 1. If you've not yet serviced the first ready channel, you now have two ready channels to service. However, only one channel became ready between these select() calls.

A set of the selected keys (the ready set) is now obtained by invoking Selector's Set<SelectionKey> selectedKeys() method. Invoke the set's iterator() method to obtain an iterator over these keys.

Note The aforementioned methods offer a convenient alternative to specifying expressions such as key.readyOps() & OP_READ != 0. SelectionKey's int readyOps() method returns the key's ready set. The returned set will only contain operation bits that are valid for this key's channel. For example, it never returns an operation bit that indicates that a read-only channel is ready for writing. Note that every selectable channel also declares an int validOps() method, which returns a bitwise ORed set of operations that are valid for the channel.
Once the application determines that a channel is ready to perform a specific operation, it can call SelectionKey's SelectableChannel channel() method to obtain the channel and then perform work on that channel.

Note SelectionKey also declares a Selector selector() method that returns the selector for which the key was created.

When you're finished processing a channel, you must remove the key from the set of keys; the selector doesn't perform this task. The next time the channel becomes ready, the Selector will add the key to the selected key set.

The following code fragment continues from the previous code fragment and demonstrates the aforementioned tasks:

```java
while (true)
{
    int numReadyChannels = selector.select();
    if (numReadyChannels == 0)
        continue; // there are no ready channels to process

    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

    while (keyIterator.hasNext())
    {
        SelectionKey key = keyIterator.next();

        if (key.isAcceptable())
        {
            // A connection was accepted by a ServerSocketChannel.
            ServerSocketChannel server = (ServerSocketChannel) key.channel();
            SocketChannel client = server.accept();
            if (client == null) // in case accept() returns null
                continue;
            client.configureBlocking(false); // must be nonblocking
            // Register socket channel with selector for read operations.
            client.register(selector, SelectionKey.OP_READ);
        }
        else
            if (key.isReadable())
            {
                // A socket channel is ready for reading.
                SocketChannel client = (SocketChannel) key.channel();
                // Perform work on the socket channel.
            }
    }
}
```
else
    if (key.isWritable())
    {
        // A socket channel is ready for writing.
        SocketChannel client = (SocketChannel) key.channel();
        // Perform work on the socket channel.
    }
    
    keyIterator.remove();
}

In addition to registering the server socket channel with the selector, each incoming client socket channel is also registered with the server socket channel. When a client socket channel becomes ready for read or write operations, key.isReadable() or key.isWritable() for the associated socket channel returns true and the socket channel can be read or written.

A key represents a relationship between a selectable channel and a selector. This relationship can be terminated by invoking SelectionKey's void cancel() method. Upon return, the key will be invalid and will have been added to its selector's cancelled-key set. The key will be removed from all of the selector's key sets during the next selection operation.

When you're finished with a selector, call Selector's void close() method. If a thread is currently blocked in one of this selector's selection methods, it's interrupted as if by invoking the selector's wakeup() method. Any uncancelled keys still associated with this selector are invalidated, their channels are deregistered, and any other resources associated with this selector are released. If this selector is already closed, invoking close() has no effect.

**Selector Demonstration**

Selectors are commonly used in server applications. Listing 13-17 presents the source code to a server application that sends its local time to clients.

**Listing 13-17. Serving Time to Clients**

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
```
public class SelectorServer
{
    final static int DEFAULT_PORT = 9999;

    static ByteBuffer bb = ByteBuffer.allocateDirect(8);

    public static void main(String[] args) throws IOException
    {
        int port = DEFAULT_PORT;
        if (args.length > 0)
            port = Integer.parseInt(args[0]);
        System.out.println("Server starting ... listening on port " + port);

        ServerSocketChannel ssc = ServerSocketChannel.open();
        ServerSocket ss = ssc.socket();
        ss.bind(new InetSocketAddress(port));
        ssc.configureBlocking(false);

        Selector s = Selector.open();
        ssc.register(s, SelectionKey.OP_ACCEPT);

        while (true)
        {
            int n = s.select();
            if (n == 0)
                continue;
            Iterator it = s.selectedKeys().iterator();
            while (it.hasNext())
            {
                SelectionKey key = (SelectionKey) it.next();
                if (key.isAcceptable())
                {
                    SocketChannel sc = ((ServerSocketChannel) key.channel()).accept();
                    if (sc == null)
                        continue;
                    System.out.println("Receiving connection");
                    bb.clear();
                    bb.putLong(System.currentTimeMillis());
                    bb.flip();
                    System.out.println("Writing current time");
                    while (bb.hasRemaining())
                        sc.write(bb);
                    sc.close();
                }
                it.remove();
            }
        }
    }
}

Listing 13-17’s server application consists of a SelectorServer class. This class allocates a direct byte buffer after this class is loaded.
When the `main()` method is executed, it first checks for a command-line argument, which is assumed to represent a port number. If no argument is specified, a default port number is used; otherwise, `main()` tries to convert it to an integer representing the port by passing the argument to `Integer.parseInt()` (Remember that this method throws `java.lang.NumberFormatException` when a noninteger argument is passed.)

After outputting a startup message that identifies the listening port, `main()` obtains a server socket channel followed by the underlying socket, which is bound to the specified port. The server socket channel is then configured for nonblocking mode in preparation for registering this channel with a selector.

A selector is now obtained and the server socket channel registers itself with the selector so that it can learn when the channel is ready to perform an accept operation. The returned key isn’t saved because it’s never canceled (and the selector is never closed).

`main()` now enters an infinite loop, first invoking the selector’s `select()` method. If the server socket channel isn’t ready (`select()` returns 0), the rest of the loop is skipped.

The selected keys (just one key) along with an iterator for iterating over them are now obtained and `main()` enters an inner loop to loop over these keys. Each key’s `isAcceptable()` method is invoked to find out if the server socket channel is ready to perform an accept operation. If this is the case, the channel is obtained and cast to `ServerSocketChannel`, and `ServerSocketChannel`’s `accept()` method is called to accept the new connection.

To guard against the unlikely possibility of the returned `SocketChannel` instance being null (accept() returns null when the server socket channel is in nonblocking mode and no connection is available to be accepted), `main()` tests for this scenario and continues the loop when null is detected.

A message about receiving a connection is output, and the byte buffer is cleared in preparation for storing the local time. After this long integer has been stored in the buffer, the buffer is flipped in preparation for draining. A message about writing the current time is output and the buffer is drained. The socket channel is then closed and the key is removed from the set of keys.

Compile Listing 13-17 as follows:

```
javac SelectorServer.java
```

Run this application as follows:

```
java SelectorServer
```

You should observe the following output and the server should continue to run:

```
Server starting ... listening on port 9999
```

We need a client to exercise this server. Listing 13-18 presents the source code to a sample client application.
Listing 13-18. Receiving Time from the Server

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.util.Date;

public class SelectorClient {
    final static int DEFAULT_PORT = 9999;

    static ByteBuffer bb = ByteBuffer.allocateDirect(8);

    public static void main(String[] args) {
        int port = DEFAULT_PORT;
        if (args.length > 0)
            port = Integer.parseInt(args[0]);

        try {
            SocketChannel sc = SocketChannel.open();
            InetSocketAddress addr = new InetSocketAddress("localhost", port);
            sc.connect(addr);

            long time = 0;
            while (sc.read(bb) != -1) {
                bb.flip();
                while (bb.hasRemaining()) {
                    time <<= 8;
                    time |= bb.get() & 255;
                }
                bb.clear();
            }
            System.out.println(new Date(time));
            sc.close();
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```
Listing 13-18 is much simpler than Listing 13-17 because selectors aren’t used. There’s no need for a selector in this simple application. You would typically use selectors in a client context when the client interacts with several servers.

There are a couple of interesting items in the source code:

- `bb.get()` returns a 32-bit integer representation of an 8-bit byte. Sign extension is used for byte values greater than 127, which are regarded as negative numbers. Because leading one bits affect the result after bitwise ORing them with `time`, they are removed by bitwise ANDing the integer with 255.

- This value in `time` is passed to the `java.util.Date(long time)` constructor when a new `Date` object is constructed. In turn, the `Date` object is passed to `System.out.println()`, which invokes `Date`'s `toString()` method to obtain a human-readable date/time string. (I discuss `Date` in Chapter 16.)

Compile Listing 13-18 as follows:

```
javac SelectorClient.java
```

In a second command window, run this application as follows:

```
java SelectorClient
```

You should observe output similar to the following:

```
Mon Jan 13 18:48:10 CST 2014
```

In the server command window, you should observe the following messages:

```
Receiving connection
Writing current time
```

### Working with Regular Expressions

Text-processing applications often need to match text against patterns (character strings that concisely describe sets of strings that are considered to be matches). For example, an application might need to locate all occurrences of a specific word pattern in a text file so that it can replace those occurrences with another word. NIO includes regular expressions to help text-processing applications perform pattern matching with high performance.

### Pattern, PatternSyntaxException, and Matcher

A regular expression (also known as a `regex` or `regexp`) is a string-based pattern that represents the set of strings that match this pattern. The pattern consists of literal characters and metacharacters, which are characters with special meanings instead of literal meanings.
The Regular Expressions API provides the `java.util.regex.Pattern` class to represent patterns via compiled regexes. Regexes are compiled for performance reasons; pattern matching via compiled regexes is much faster than if the regexes were not compiled. Table 13-3 describes `Pattern`'s methods.

**Table 13-3. Pattern Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static Pattern compile(String regex)</code></td>
<td>Compiles regex and returns its <code>Pattern</code> object. This method throws <code>java.util.regex.PatternSyntaxException</code> when regex's syntax is invalid.</td>
</tr>
<tr>
<td><code>static Pattern compile(String regex, int flags)</code></td>
<td>Compiles regex according to the given flags (a bitset consisting of some combination of <code>Pattern</code>'s <code>CANON_EQ</code>, <code>CASE_INSENSITIVE</code>, <code>COMMENTS</code>, <code>DOTALL</code>, <code>LITERAL</code>, <code>MULTILINE</code>, <code>UNICODE_CASE</code>, and <code>UNIX_LINES</code> constants) and returns its <code>Pattern</code> object. This method throws <code>PatternSyntaxException</code> when regex's syntax is invalid, and throws <code>IllegalArgumentException</code> when bit values other than those corresponding to the defined match flags are set in <code>flags</code>.</td>
</tr>
<tr>
<td><code>int flags()</code></td>
<td>Returns this <code>Pattern</code> object's match flags. This method returns 0 for <code>Pattern</code> instances created via <code>compile(String)</code> and the bitset of flags for <code>Pattern</code> instances created via <code>compile(String, int)</code>.</td>
</tr>
<tr>
<td><code>Matcher matcher(CharSequence input)</code></td>
<td>Returns a <code>java.util.regex.Matcher</code> that will match <code>input</code> against this <code>Pattern</code>'s compiled regex.</td>
</tr>
<tr>
<td><code>static boolean matches(String regex, CharSequence input)</code></td>
<td>Compiles regex and attempts to match <code>input</code> against the compiled regex. Returns true when there is a match; otherwise, returns false. This convenience method is equivalent to <code>Pattern.compile(regex).matcher(input).matches()</code> and throws <code>PatternSyntaxException</code> when regex's syntax is invalid.</td>
</tr>
<tr>
<td><code>String pattern()</code></td>
<td>Returns this <code>Pattern</code>'s uncompiled regex.</td>
</tr>
<tr>
<td><code>static String quote(String s)</code></td>
<td>Quotes s using “\Q” and “\E” so that all other metacharacters lose their special meaning. When the returned <code>java.lang.String</code> object is later compiled into a <code>Pattern</code> instance, it only can be matched literally.</td>
</tr>
<tr>
<td><code>String[] split(CharSequence input)</code></td>
<td>Splits <code>input</code> around matches of this <code>Pattern</code>'s compiled regex and returns an array containing the matches.</td>
</tr>
<tr>
<td><code>String[] split(CharSequence input, int limit)</code></td>
<td>Splits <code>input</code> around matches of this <code>Pattern</code>'s compiled regex; <code>limit</code> controls the number of times the compiled regex is applied and thus affects the length of the resulting array.</td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td>Returns this <code>Pattern</code>'s uncompiled regex.</td>
</tr>
</tbody>
</table>

Table 13-3 reveals the `java.lang.CharSequence` interface, which describes a readable and immutable sequence of `char` values; the underlying implementation may be mutable. Instances of any class that implements this interface (such as `String`, `java.lang.StringBuffer`, and `java.lang.StringBuilder`) can be passed to `Pattern` methods that take `CharSequence` arguments (such as `split(CharSequence)`).
Table 13-3 also reveals that each of Pattern's compile() methods and its matches() method (which calls the compile(String) method) throws PatternSyntaxException when a syntax error is encountered while compiling the pattern argument. Table 13-4 describes PatternSyntaxException's methods.

**Table 13-4. PatternSyntaxException Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getDescription()</td>
<td>Returns a description of the syntax error.</td>
</tr>
<tr>
<td>int getIndex()</td>
<td>Returns the approximate index of where the syntax error occurred in the pattern or -1 when the index isn't known.</td>
</tr>
<tr>
<td>String getMessage()</td>
<td>Returns a multiline string containing the description of the syntax error and its index, the erroneous pattern, and a visual indication of the error index within the pattern.</td>
</tr>
<tr>
<td>String getPattern()</td>
<td>Returns the erroneous pattern.</td>
</tr>
</tbody>
</table>

Finally, Table 13-4's Matcher matcher(CharSequence input) method reveals that the Regular Expressions API also provides the Matcher class, whose matchers attempt to match compiled regexes against input text. Matcher declares the following methods to perform matching operations:

- **boolean matches()**: Attempts to match the entire region against the pattern. When the match succeeds, more information can be obtained by calling Matcher's start(), end(), and group() methods. For example, int start() returns the start index of the previous match, int end() returns the offset of the first character following the previous match, and String group() returns the input subsequence matched by the previous match. Each method throws java.lang.IllegalStateException when a match has not yet been attempted or the previous match attempt failed.

- **boolean lookingAt()**: Attempts to match the input sequence, starting at the beginning of the region, against the pattern. As with matches(), this method always starts at the beginning of the region. Unlike matches(), lookingAt() doesn't require that the entire region be matched. When the match succeeds, more information can be obtained by calling Matcher's start(), end(), and group() methods.

- **boolean find()**: Attempts to find the next subsequence of the input sequence that matches the pattern. It starts at the beginning of this matcher's region, or, if a previous call to this method was successful and the matcher hasn't since been reset (by calling Matcher's Matcher reset() or Matcher reset(CharSequence input) method), at the first character not matched by the previous match. When the match succeeds, more information can be obtained by calling Matcher's start(), end(), and group() methods.
I've created a simple application that demonstrates `Pattern`, `PatternSyntaxException`, and `Matcher`. Listing 13-19 presents this application's source code.

Listing 13-19. Playing with Regular Expressions

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class RegExDemo
{
    public static void main(String[] args)
    {
        if (args.length != 2)
        {
            System.err.println("usage: java RegExDemo regex input");
            return;
        }
        try
        {
            System.out.println("regex = " + args[0]);
            System.out.println("input = " + args[1]);
            Pattern p = Pattern.compile(args[0]);
            Matcher m = p.matcher(args[1]);
            while (m.find())
            {
                System.out.println("Located [" + m.group() + "] starting at "
                    + m.start() + " and ending at " + (m.end() - 1));
            }
        } catch (PatternSyntaxException pse)
        {
            System.err.println("Bad regex: " + pse.getMessage());
            System.err.println("Description: " + pse.getDescription());
            System.err.println("Index: " + pse.getIndex());
            System.err.println("Incorrect pattern: " + pse.getPattern());
        }
    }
}
```

Compile Listing 13-19 as follows:

```
javac RegExDemo.java
```
Run this application as follows:

```java
java RegExDemo ox ox
```

You’ll discover the following output:

```java
regex = ox
input = ox
Located [ox] starting at 0 and ending at 1
```

`find()` searches for a match by comparing regex characters with the input characters in left-to-right order and returns true because o equals o and x equals x.

Continue by executing the following command:

```java
java RegExDemo box ox
```

This time, you’ll discover the following output:

```java
regex = box
input = ox
```

`find()` first compares regex character b with input character o. Because these characters are not equal and because there are not enough characters in the input to continue the search, `find()` doesn’t output a “Located” message to indicate a match.

However, if you execute `java RegExDemo ox box`, you’ll discover a match:

```java
regex = ox
input = box
Located [ox] starting at 1 and ending at 2
```

The ox regex consists of literal characters. More sophisticated regexes combine literal characters with metacharacters (such as the period [.]) and other regex constructs.

**Tip** To specify a metacharacter as a literal character, precede the metacharacter with a backslash character (as in \\.) or place the metacharacter between \Q and \E (as in \Q.\E). In either case, make sure to double the backslash character when the escaped metacharacter appears in a string literal, such as "\\." or "\\Q.\\E".

The period metacharacter matches all characters except for the line terminator. For example, each of `java RegExDemo .ox box` and `java RegExDemo .ox fox` report a match because the period matches the b in box and the f in fox.
Character Classes

A character class is a set of characters appearing between [ and ]. There are six kinds of character classes:

- A simple character class consists of literal characters placed side by side and matches only these characters. For example, [abc] consists of characters a, b, and c. Also, \texttt{java RegExDemo t[aiou]ck} tack reports a match because a is a member of \texttt{[aiou]}. It also reports a match when the input is tick, tock, or tuck because i, o, and u are members.

- A negation character class consists of a circumflex metacharacter (^), followed by literal characters placed side by side, and it matches all characters except for those in the class. For example, \[^abc\] consists of all characters except for a, b, and c. Also, \texttt{java RegExDemo "[^b]\0x" box} doesn’t report a match because b isn’t a member of \[^b\], whereas \texttt{java RegExDemo "[^b]\0x" fox} reports a match because f is a member. (The double quotes surrounding \[^b\]0x are necessary on my Windows 7 platform because ^ is treated specially at the command line."

- A range character class consists of successive literal characters expressed as a starting literal character, followed by the hyphen metacharacter (-), followed by an ending literal character, and matches all characters in this range. For example, \[a-z\] consists of all characters from a through z. Also, \texttt{java RegExDemo [h-l]ouse} house reports a match because h is a member of the class, whereas \texttt{java RegExDemo [h-l]ouse mouse} doesn’t report a match because m lies outside of the range and is therefore not part of the class. You can combine multiple ranges within the same range character class by placing them side by side; for example, \[A-Za-z\] consists of all uppercase and lowercase Latin letters.

- A union character class consists of multiple nested character classes and matches all characters that belong to the resulting union. For example, \[abc[u-z]\] consists of characters a, b, c, u, v, w, x, y, and z. Also, \texttt{java RegExDemo \[0-9\][A-F][a-f]\}} e reports a match because e is a hexadecimal character. (I could have alternatively expressed this character class as \[0-9A-Fa-f\] by combining multiple ranges.)

- An intersection character class consists of multiple &&–separated nested character classes and matches all characters that are common to these nested character classes. For example, \[a-c&&[c-f]\] consists of character c, which is the only character common to \[a-c\] and \[c-f\]. Also, \texttt{java RegExDemo "[aeiouy&&[y]]" y} reports a match because y is common to classes [aeiouy] and [y].
A *subtraction character class* consists of multiple &amp;-separated nested character classes, where at least one nested character class is a negation character class, and it matches all characters except for those indicated by the negation character class/classes. For example, `[a-z&&[^x-z]]` consists of characters a through w. (The square brackets surrounding ^x-z are necessary; otherwise, ^ is ignored and the resulting class consists of only x, y, and z.) Also, `java RegExDemo "[a-z&&[^aeiou]]" g` reports a match because g is a consonant and only consonants belong to this class. (I'm ignoring y, which is sometimes regarded as a consonant and sometimes regarded as a vowel.)

A *predefined character class* is a regex construct for a commonly specified character class. Table 13-5 identifies Pattern's predefined character classes.

<table>
<thead>
<tr>
<th>Predefined Character Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Matches any digit character. \d is equivalent to [0-9].</td>
</tr>
<tr>
<td>\D</td>
<td>Matches any nondigit character. \D is equivalent to [^\d].</td>
</tr>
<tr>
<td>\s</td>
<td>Matches any whitespace character. \s is equivalent to [\t\n\x0B\f\r\ ].</td>
</tr>
<tr>
<td>\S</td>
<td>Matches any nonwhitespace character. \S is equivalent to [^\s].</td>
</tr>
<tr>
<td>\w</td>
<td>Matches any word character. \w is equivalent to [a-zA-Z0-9].</td>
</tr>
<tr>
<td>\W</td>
<td>Matches any nonword character. \W is equivalent to [^\w].</td>
</tr>
</tbody>
</table>

For example, the following command reports a match because \w matches the word character a in abc:

```
java RegExDemo \wbc abc
```

**Capturing Groups**

A *capturing group* saves a match’s characters for later recall during pattern matching and is expressed as a character sequence surrounded by parentheses metacharacters ( and ). All characters within a capturing group are treated as a unit. For example, the (Android) capturing group combines A, n, d, r, o, i, and d into a unit. It matches the Android pattern against all occurrences of Android in the input. Each match replaces the previous match’s saved Android characters with the next match’s Android characters.

Capturing groups can appear inside other capturing groups. For example, capturing groups (A) and (B(C)) appear inside capturing group (((A)(B(C)))), and capturing group (C) appears inside capturing group (B(C)). Each nested or nonnested capturing group receives its own number, numbering starts at 1, and capturing groups are numbered from left to right. For example, (((A) (B(C)))) is assigned 1, (A) is assigned 2, (B(C)) is assigned 3, and (C) is assigned 4.
A capturing group saves its match for later recall via a *back reference*, which is a backslash character followed by a digit character denoting a capturing group number. The back reference causes the matcher to use the back reference’s capturing group number to recall the capturing group’s saved match and then use that match’s characters to attempt a further match. The following example uses a back reference to determine if the input consists of two consecutive Android patterns:

```java
RegExDemo "(Android) \1" "Android Android"
```

RegExDemo reports a match because the matcher detects Android, followed by a space, followed by Android in the input.

**Boundary Matchers and Zero-Length Matches**

A *boundary matcher* is a regex construct for identifying the beginning of a line, a word boundary, the end of text, and other commonly occurring boundaries. See Table 13-6.

<table>
<thead>
<tr>
<th>Boundary Matcher</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the beginning of the line.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of the line.</td>
</tr>
<tr>
<td>\b</td>
<td>Matches a word boundary.</td>
</tr>
<tr>
<td>\B</td>
<td>Matches a nonword boundary.</td>
</tr>
<tr>
<td>\A</td>
<td>Matches the beginning of text.</td>
</tr>
<tr>
<td>\G</td>
<td>Matches the end of the previous match.</td>
</tr>
<tr>
<td>\Z</td>
<td>Matches the end of text except for line terminator (when present).</td>
</tr>
<tr>
<td>\z</td>
<td>Matches the end of text.</td>
</tr>
</tbody>
</table>

Consider the following example:

```java
RegExDemo \b\b "I think"
```

This example reports several matches, as revealed in the following output:

```java
regex = \b\b
input = I think
Located [ ] starting at 0 and ending at -1
Located [ ] starting at 1 and ending at 0
Located [ ] starting at 2 and ending at 1
Located [ ] starting at 7 and ending at 6
```
This output reveals several zero-length matches. When a zero-length match occurs, the starting and ending indexes are equal, although the output shows the ending index to be one less than the starting index because I specified end( ) - 1 in Listing 13-19 (so that a match’s end index identifies a non-zero-length match’s last character, not the character following the non-zero-length match’s last character).

**Note** A zero-length match occurs in empty input text, at the beginning of input text, after the last character of input text, or between any two characters of that text. Zero-length matches are easy to identify because they always start and end at the same index position.

**Quantifiers**

The final regex construct I present is the quantifier, a numeric value implicitly or explicitly bound to a pattern. Quantifiers are categorized as greedy, reluctant, or possessive:

- A **greedy quantifier** (?, *, or +) attempts to find the longest match. Specify X? to find one or no occurrences of X; X* to find zero or more occurrences of X; X+ to find one or more occurrences of X; X{n} to find n occurrences of X; X{n,} to find at least n (and possibly more) occurrences of X; and X{n,m} to find at least n but no more than m occurrences of X.

- A **reluctant quantifier** (??, *?, or +?) attempts to find the shortest match. Specify X?? to find one or no occurrences of X; X*?? to find zero or more occurrences of X; X+?? to find one or more occurrences of X; X{n}?? to find n occurrences of X; X{n,}?? to find at least n (and possibly more) occurrences of X; and X{n,m}?? to find at least n but no more than m occurrences of X.

- A **possessive quantifier** (??+, *+, or ++) is similar to a greedy quantifier except that a possessive quantifier only makes one attempt to find the longest match, whereas a greedy quantifier can make multiple attempts. Specify X?+ to find one or no occurrences of X; X*+ to find zero or more occurrences of X; X++ to find one or more occurrences of X; X{n}+ to find n occurrences of X; X{n,}+ to find at least n (and possibly more) occurrences of X; and X{n,m}+ to find at least n but no more than m occurrences of X.

For an example of a greedy quantifier, execute the following command:

```java
java RegExDemo .*end "wend rend end"
```

You’ll discover the following output:

```plaintext
regex = .*end
input = wend rend end
Located [wend rend end] starting at 0 and ending at 12
```
The greedy quantifier (.*⁻¹) matches the longest sequence of characters that terminates in end. It starts by consuming all of the input text and then is forced to back off until it discovers that the input text terminates with these characters.

For an example of a reluctant quantifier, execute the following command:

```
java RegExDemo .*?end "wend rend end"
```

You’ll discover the following output:

```
regex = .*?end
input = wend rend end
Located [wend] starting at 0 and ending at 3
Located [ rend] starting at 4 and ending at 8
Located [ end] starting at 9 and ending at 12
```

The reluctant quantifier (.*⁻¹) matches the shortest sequence of characters that terminates in end. It begins by consuming nothing and then slowly consumes characters until it finds a match. It then continues until it exhausts the input text.

For an example of a possessive quantifier, execute the following command:

```
java RegExDemo .*+end "wend rend end"
```

You’ll discover the following output:

```
regex = .*+end
input = wend rend end
```

The possessive quantifier (.*⁻¹) doesn’t detect a match because it consumes the entire input text, leaving nothing left over to match end at the end of the regex. Unlike a greedy quantifier, a possessive quantifier doesn’t back off.

While working with quantifiers, you’ll probably encounter zero-length matches. For example, execute the following command:

```
java RegExDemo 1? 101101
```

You should observe the following output:

```
regex = 1?
input = 101101
Located [1] starting at 0 and ending at 0
Located [ ] starting at 1 and ending at 0
Located [1] starting at 2 and ending at 2
Located [1] starting at 3 and ending at 3
Located [ ] starting at 4 and ending at 3
Located [1] starting at 5 and ending at 5
Located [ ] starting at 6 and ending at 5
```
The result of this greedy quantifier is that 1 is detected at locations 0, 2, 3, and 5 in the input text, and that nothing is detected (a zero-length match) at locations 1, 4, and 6.

This time, execute the following command:

```java
java RegExDemo 1?? 101101
```

You should observe the following output:

```
regex = 1??
input = 101101
Located [] starting at 0 and ending at -1
Located [] starting at 1 and ending at 0
Located [] starting at 2 and ending at 1
Located [] starting at 3 and ending at 2
Located [] starting at 4 and ending at 3
Located [] starting at 5 and ending at 4
Located [] starting at 6 and ending at 5
```

This output might look surprising, but remember that a reluctant quantifier looks for the shortest match, which (in this case) is no match at all.

Finally, execute the following command:

```java
java RegExDemo 1+? 101101
```

You should observe the following output:

```
regex = 1+?
input = 101101
Located [1] starting at 0 and ending at 0
Located [1] starting at 2 and ending at 2
Located [1] starting at 3 and ending at 3
Located [1] starting at 5 and ending at 5
```

This possessive quantifier only matches the locations where 1 is detected in the input text. It doesn’t perform zero-length matches.

---

**Note** Check out the Java documentation on the `Pattern` class to learn about additional regex constructs.
Practical Regular Expressions

Most of the previous regex examples haven’t been practical, except to help you grasp how to use the various regex constructs. In contrast, the following examples reveal a regex that matches phone numbers of the form (ddd) ddd-dddd or ddd-dddd. A single space appears between (ddd) and ddd; there’s no space on either side of the hyphen.

```java
RegExDemo "\(\d{3}\)?\s*\d{3}-\d{4}" "(800) 555-1212"
regex = \(\(\d{3}\)\)?\s*\d{3}-\d{4}
input = (800) 555-1212
Located [(800) 555-1212] starting at 0 and ending at 13
```

```java
RegExDemo "\(\d{3}\)?\s*\d{3}-\d{4}" 555-1212
regex = \(\(\d{3}\)\)?\s*\d{3}-\d{4}
input = 555-1212
Located [555-1212] starting at 0 and ending at 7
```


Working with Charsets

In Chapter 11, I briefly introduced the concepts of character set and character encoding. I also referred to some of the types located in the java.nio.charset package. In this section, I expand on these topics and explore this package in more detail. I also briefly revisit the String class, discussing that part of String that’s relevant to the discussion.

A Brief Review of the Fundamentals

Java uses Unicode to represent characters. (Unicode is a 16-bit character set standard [actually, more of an encoding standard because some characters are represented by multiple numeric values; each value is known as a code point] whose goal is to map all of the world’s significant character sets into an all-encompassing mapping). Although Unicode makes it much easier to work with characters from different languages, it doesn’t automate everything and you often need to work with charsets. Before I dig into this topic, you should understand the following terms:

- **Character**: A meaningful symbol. For example, “$” and “E” are characters. These symbols predate the computer era.

- **Character set**: A set of characters. For example, uppercase letters A through Z could be considered to form a character set. No numeric values are assigned to the characters in the set. There is no relationship to Unicode, ASCII, EBCDIC, or any other kind of character set standard.

- **Coded character set**: A character set where each character is assigned a unique numeric value. Standards bodies such as US-ASCII or ISO-8859-1 define mappings from characters to numeric values.
Character-encoding scheme: An encoding of a coded character set's numeric values to sequences of bytes that represent these values. Some encodings are one-to-one. For example, in ASCII, character A is mapped to integer 65 and encoded as integer 65. For some other mappings, encodings are one-to-one or one-to-many. For example, UTF-8 encodes Unicode characters. Each character whose numeric value is less than 128 is encoded as a single byte to be compatible with ASCII. Other Unicode characters are encoded as 2-to-6-byte sequences. See www.ietf.org/rfc/rfc2279.txt for more information.

Charset: A coded character set combined with a character-encoding scheme.Charsets are described by the abstract java.nio.charset.CharSet class.

Although Unicode is widely used and increasing in popularity, other character set standards are also used. Because operating systems perform I/O at the byte level, and because files store data as byte sequences, it's necessary to translate between byte sequences and the characters that are encoded into these sequences. Charset and the other classes located in the java.nio.charset package address this translation task.

Working withCharsets

Beginning with JDK 1.4, virtual machines were required to support a standard collection of charsets and could support additional charsets. They also support the default charset, which doesn’t have to be one of the standard charsets and is obtained when the virtual machine starts running. Table 13-7 identifies and describes the standard charsets.

<table>
<thead>
<tr>
<th>Charset Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>US-ASCII</td>
<td>The 7-bit ASCII that forms the American English character set. Also known as the basic Latin block in Unicode.</td>
</tr>
<tr>
<td>ISO-8859-1</td>
<td>The 8-bit character set used by most European languages. It’s a superset of ASCII and includes most non-English European characters.</td>
</tr>
<tr>
<td>UTF-8</td>
<td>An 8-bit byte-oriented character encoding for Unicode. Characters are encoded in 1 to 6 bytes.</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>A 16-bit encoding using big-endian order for Unicode. Characters are encoded in 2 bytes with the high-order 8 bits written first.</td>
</tr>
<tr>
<td>UTF-16LE</td>
<td>A 16-bit encoding using little-endian order for Unicode. Characters are encoded in 2 bytes with the low-order 8 bits written first.</td>
</tr>
<tr>
<td>UTF-16</td>
<td>A 16-bit encoding whose endian order is determined by an optional byte-order mark.</td>
</tr>
</tbody>
</table>

Charset names are case-insensitive and are maintained by the Internet Assigned Names Authority (IANA). The names in Table 13-7 are included in IANA’s official registry.

UTF-16BE and UTF-16LE encode each character as a 2-byte sequence in big-endian or little-endian order, respectively. A decoder for a UTF-16BE- or UTF-16LE-encoded byte sequence needs to know how the byte sequence was encoded. In contrast, UTF-16 relies on a byte order mark that appears at the beginning of the sequence. If this mark is absent, decoding proceeds according to UTF-16BE.
(Java's native byte order). If this mark equals \uFEFF, the sequence is decoded according to UTF-16BE. If this mark equals \uFFFE, the sequence is decoded according to UTF-16LE.

Each charset name is associated with a Charset object, which you obtain by invoking one of this class's factory methods. Listing 13-20 presents an application that shows you how to use this class to obtain the default and standard charsets, which are then used to encode characters into byte sequences.

Listing 13-20. Using Charsets to Encode Characters into Byte Sequences

```java
import java.nio.ByteBuffer;
import java.nio.charset.Charset;

public class CharsetDemo {
    public static void main(String[] args) {
        String msg = "façade touché";
        String[] csNames = {
            "US-ASCII",
            "ISO-8859-1",
            "UTF-8",
            "UTF-16BE",
            "UTF-16LE",
            "UTF-16"
        };

        encode(msg, Charset.defaultCharset());
        for (String csName: csNames)
            encode(msg, Charset.forName(csName));
    }

    static void encode(String msg, Charset cs) {
        System.out.println("Charset: ", cs.toString());
        System.out.println("Message: ", msg);

        ByteBuffer buffer = cs.encode(msg);
        System.out.println("Encoded: ");

        for (int i = 0; buffer.hasRemaining(); i++) {
            int _byte = buffer.get() & 255;
            char ch = (char) _byte;
            if (Character.isWhitespace(ch) || Character.isISOControl(ch))
                ch = '\0000';
            System.out.printf("%2d: %02x (%c)%n", i, _byte, ch);
        }
    }
}
```
Listing 13-20’s `main()` method first creates a message consisting of two French words and an array of names for the standard collection of charsets. Next, it invokes the `encode()` method to encode the message according to the default charset, which it obtains by calling Charset’s `Charset defaultCharset()` factory method. Continuing, `main()` invokes `encode()` for each of the standard charsets. Charset’s `Charset forName(String charsetName)` factory method is used to obtain the Charset instance that corresponds to `charsetName`.

**Caution**  `forName()` throws `java.nio.charset.IllegalArgumentException` when the specified charset name is illegal and throws `java.nio.charset.UnsupportedEncodingException` when the desired charset isn’t supported by the virtual machine.

The `encode()` method first identifies the charset and the message. It then invokes Charset’s `ByteBuffer encode(String s)` method to return a new `ByteBuffer` object containing the bytes that encode the characters from `s`.

`main()` next iterates over the bytes in the byte buffer, converting each byte to a character. It uses `java.lang.Character`’s `isWhitespace()` and `isISOControl()` methods to determine if the character is whitespace or a control character (neither is regarded as printable) and converts such a character to Unicode 0 (empty string). (A carriage return or newline would screw up the output, for example.) Finally, the index of the character, its hexadecimal value, and the character itself are printed to the standard output stream. I chose to use `System.out.printf()` for this task. You’ll learn about this method in the next section.

Compile Listing 13-20 as follows:

```
javac CharsetDemo.java
```

Run the application as follows:

```
java CharsetDemo
```

You should observe the following output:

```
Charset: windows-1252
Message: façade touché
Encoded:
  0: 66 (f)
  1: 61 (a)
  2: e7 (ç)
  3: 61 (a)
  4: 64 (d)
  5: 65 (e)
  6: 20 ( )
  7: 74 (t)
  8: 6f (o)
```
9: 75 (u)
10: 63 (c)
11: 68 (h)
12: e9 (é)

Charset: US-ASCII
Message: façade touché
Encoded:
0: 66 (f)
1: 61 (a)
2: 3f (?)
3: 61 (a)
4: 64 (d)
5: 65 (e)
6: 20 ( )
7: 74 (t)
8: 6f (o)
9: 75 (u)
10: 63 (c)
11: 68 (h)
12: 3f (?)

Charset: ISO-8859-1
Message: façade touché
Encoded:
0: 66 (f)
1: 61 (a)
2: e7 (ç)
3: 61 (a)
4: 64 (d)
5: 65 (e)
6: 20 ( )
7: 74 (t)
8: 6f (o)
9: 75 (u)
10: 63 (c)
11: 68 (h)
12: e9 (é)

Charset: UTF-8
Message: façade touché
Encoded:
0: 66 (f)
1: 61 (a)
2: c3 (Ã)
3: a7 ($)
4: 61 (a)
5: 64 (d)
6: 65 (e)
7: 20 ( )
8: 74 (t)
9: 6f (o)
10: 75 (u)
11: 63 (c)
12: 68 (h)
13: c3 (Å)
14: a9 (©)

Charset: UTF-16BE
Message: façade touché
Encoded:
0: 00 ( )
1: 66 (f)
2: 00 ( )
3: 61 (a)
4: 00 ( )
5: e7 (ç)
6: 00 ( )
7: 61 (a)
8: 00 ( )
9: 64 (d)
10: 00 ( )
11: 65 (e)
12: 00 ( )
13: 20 ( )
14: 00 ( )
15: 74 (t)
16: 00 ( )
17: 6f (o)
18: 00 ( )
19: 75 (u)
20: 00 ( )
21: 63 (c)
22: 00 ( )
23: 68 (h)
24: 00 ( )
25: e9 (é)

Charset: UTF-16LE
Message: façade touché
Encoded:
0: 66 (f)
1: 00 ( )
2: 61 (a)
3: 00 ( )
4: e7 (ç)
5: 00 ( )
6: 61 (a)
7: 00 ( )
8: 64 (d)
9: 00 ( )
10: 65 (e)
11: 00 ( )
12: 20 ( )
In addition to providing encoding methods such as the aforementioned `ByteBuffer encode(String s)` method, `Charset` provides a complementary `CharBuffer decode(ByteBuffer buffer)` decoding method. The return type is `CharBuffer` because byte sequences are decoded into characters.
Note ByteBuffer encode(String s) is a convenience method for specifying CharBuffer.wrap(s) and passing the result to the ByteBuffer encode(CharBuffer buffer) method.

If you dig deeper into Charset, you’ll encounter the following pair of methods:

- CharsetEncoder newEncoder()
- CharsetDecoder newDecoder()

These methods perform the actual work of encoding and decoding. Charset’s encode() and decode() methods delegate to the java.nio.charset.CharsetEncoder and java.nio.CharsetDecoder objects returned from newEncoder() and newDecoder() and invoke their encode() and decode() (along with additional) methods. (For brevity, I don’t discuss CharsetEncoder and CharsetDecoder.)

Charsets and the String Class

The String class describes a string as a sequence of characters. It declares constructors that can be passed byte arrays. Because a byte array contains an encoded character sequence, a charset is required to decode them. The following is a partial list of String constructors that work with charsets:

- String(byte[] data): Constructs a new String instance by decoding the specified array of bytes using the platform’s default charset.
- String(byte[] data, int offset, int byteCount): Constructs a new String instance by decoding the specified subsequence of the byte array using the platform’s default charset.
- String(byte[] data, String charsetName): Constructs a new String instance by decoding the specified array of bytes using the named charset.

Furthermore, String declares methods that encode its sequence of characters into a byte array with help from the default charset or a named charset. Two of these methods are the following:

- byte[] getBytes(): Returns a new byte array containing the characters of this string encoded using the platform’s default charset.
- byte[] getBytes(String charsetName): Returns a new byte array containing the characters of this string encoded using the named charset.

Note that String(byte[] data, String charsetName) and byte[] getBytes(String charsetName) throw java.io.UnsupportedEncodingException when the charset isn’t supported.

I’ve created a small application that demonstrates String and charsets. Listing 13-21 presents the source code.
Listing 13-21. Using Charsets with String

```java
import java.io.UnsupportedEncodingException;

public class CharsetDemo
{
    public static void main(String[] args) throws UnsupportedEncodingException
    {
        byte[] encodedMsg =
        {
            0x66, 0x61, (byte) 0xc3, (byte) 0xa7, 0x61, 0x64, 0x65, 0x20, 0x74,
            0x6f, 0x75, 0x63, 0x68, (byte) 0xc3, (byte) 0xa9
        };
        String s = new String(encodedMsg, "UTF-8");
        System.out.println(s);
        System.out.println();
        byte[] bytes = s.getBytes();
        for (byte _byte: bytes)
            System.out.print(Integer.toHexString(_byte & 255) + " ");
        System.out.println();
    }
}
```

Listing 13-21’s `main()` method first creates a byte array containing a UTF-8 encoded message. It then converts this array to a String object via the UTF-8 charset. After outputting the resulting String object, it extracts this object’s bytes into a new byte array and proceeds to output these bytes in hexadecimal format. As demonstrated earlier in this chapter, I bitwise AND each byte value with 255 to remove the 0xFF sign extension bytes for negative integers when the 8-bit byte integer value is converted to a 32-bit integer value. These sign extension bytes would otherwise be output.

Compile Listing 13-21 (`javac CharsetDemo.java`), and run this application (`java CharsetDemo`). You should observe the following output:

```
façade touché
66 61 e7 61 64 65 20 74 6f 75 63 68 e9
```

You might be wondering why you observe `e7` instead of `c3 a7` (Latin small letter c with a cedilla [hook or tail]) and `e9` instead of `c3 a9` (Latin small letter e with an acute accent). The answer is that I invoked the no-argument `getBytes()` method to encode the string. This method uses the default charset, which is `windows-1252` on my platform. According to this charset, `e7` is equivalent to `c3 a7` and `e9` is equivalent to `c3 a9`. The result is a shorter encoded sequence.

### Working with Formatter and Scanner

The description for JSR 51 ([http://jcp.org/en/jsr/detail?id=51](http://jcp.org/en/jsr/detail?id=51)) indicates that a simple printf-style formatting facility was proposed for inclusion in NIO. If you’re familiar with the C language, you’ve probably worked with the `printf()` family of functions that support formatted output. You’ve also probably worked with the complementary `scanf()` family that support formatted input.
One feature that makes the `printf()` and `scanf()` functions useful is varargs, which lets you pass a variable number of arguments to these functions. Because support for varargs wasn’t added to Java until JDK 1.5, and because this support is very useful for achieving formatted output (Java doesn’t need it for formatted input), the formatted `printf`-style formatting facility was deferred to JDK 1.5.

**Working with Formatter**

Java 5 introduced the `java.util.Formatter` class as an interpreter for `printf()`-style format strings. This class provides support for layout justification and alignment; common formats for numeric, string, and date/time data; and more. Commonly used Java types (such as byte and `BigDecimal`) are supported. Also, limited formatting customization for arbitrary user-defined types is provided through the associated `java.util.Formattable` interface and `java.util.FormattableFlags` class.

`Formatter` declares several constructors for creating `Formatter` objects. These constructors let you specify where you want formatted output to be sent. For example, `Formatter()` writes formatted output to an internal `StringBuilder` instance and `Formatter(OutputStream os)` writes formatted output to the specified output stream. You can access the destination by calling `Formatter`’s `Appendable out()` method.

**Note** The `java.lang.Appendable` interface describes an object to which `char` values and character sequences can be appended. Classes (such as `StringBuilder`) whose instances are to receive formatted output (via the `Formatter` class) implement `Appendable`. This interface declares methods such as `Appendable append(char c)`, which appends `c`’s character to this appendable. When an I/O error occurs, this method throws `IOException`.

After creating a `Formatter` object, you would call a `format()` method to format a varying number of values. For example, `Formatter format(String format, Object... args)` formats the `args` array according to the string of format specifiers passed to the `format` parameter, and it returns a reference to the invoking `Formatter` so that you can chain `format()` calls together (for convenience).

Each format specifier has one of the following syntaxes:

- `%[argument_index$][flags][width][.precision]conversion`
- `%[argument_index$][flags][width]conversion`
- `%[flags][width]conversion`

The first syntax describes a format specifier for general, character, and numeric types. The second syntax describes a format specifier for types that are used to represent dates and times. The third syntax describes a format specifier that doesn’t correspond to arguments.

The optional `argument_index` is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by `1$`, the second argument is referenced by `2$`, and so on.

The optional `flags` are a set of characters that modify the output format. The set of valid flags depends on the conversion.
The optional *width* is a positive decimal integer indicating the minimum number of characters to be written to the output.

The optional *precision* is a nonnegative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.

The required conversion depends on the syntax. For the first syntax, it's a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type. For the second syntax, it's a two-character sequence. The first character is *t* or *T*. The second character indicates the format to be used. For the third syntax, it's a character indicating content to be inserted in the output.

Conversions are divided into six categories: general, character, numeric (integer or floating-point), date/time, percent, and line separator. The following list identifies a few example format specifiers and their conversions:

- `%d`: Formats argument as a decimal integer.
- `%x`: Formats argument as a hexadecimal integer.
- `%c`: Formats argument as a character.
- `%f`: Formats argument as a decimal number.
- `%s`: Formats argument as a string.
- `%n`: Outputs a platform-specific line separator.
- `%10.2f`: Formats argument as a decimal number with 10 as the minimum number of characters to be written (leading spaces are written when the number is smaller than the width) and 2 as the number of characters to be written after the decimal point.
- `%05d`: Formats argument as a decimal integer with 5 as the minimum number of characters to be written (leading 0s are written when the number is smaller than the width).

When you’re finished with the formatter, you might want to invoke the void *flush()* method to ensure that any buffered output in the destination is written to the underlying stream. You would typically invoke *flush()* when the destination is a file.

Continuing, invoke the formatter’s void *close()* method. In addition to closing the formatter, this method also closes the underlying output destination when this destination’s class implements the java.io.Closeable interface. If the formatter has been closed, this method has no effect. Attempting to format after calling *close()* results in java.util.FormatterClosedException.

Listing 13-22 provides a simple demonstration of *Formatter* using the aforementioned format specifiers.
Listing 13-22. Demonstrating the Formatter Class

import java.util.Formatter;

public class FormatterDemo
{
   public static void main(String[] args)
   {
      Formatter formatter = new Formatter();
      formatter.format("%d", 123);
      System.out.println(formatter.toString());
      formatter.format("%x", 123);
      System.out.println(formatter.toString());
      formatter.format("%c", 'X');
      System.out.println(formatter.toString());
      formatter.format("%f", 0.1);
      System.out.println(formatter.toString());
      formatter.format("%s%n", "Hello, World");
      System.out.println(formatter.toString());
      formatter.format("%10.2f", 98.375);
      System.out.println(formatter.toString());
      formatter.format("%05d", 123);
      System.out.println(formatter.toString());
      formatter.format("%1$d %1$d", 123);
      System.out.println(formatter.toString());
      formatter.format("%d %d", 123);
      System.out.println(formatter.toString());
      formatter.close();
   }
}

Listing 13-22's main() method first creates a Formatter object via the Formatter() constructor, which sends formatted output to a StringBuilder instance. It then demonstrates the aforementioned format specifiers by invoking a format() method, followed by the toString() method to obtain the formatted content, which is subsequently output.

The formatter.format("%1$d %1$d", 123); method call accesses the single data item argument to be formatted (123) twice by referencing this argument via 1$. Without this reference, which is demonstrated via formatter.format("%d %d", 123);, an exception would be thrown because there must be a separate argument for each format specifier unless you use an argument index.

Lastly, the formatter is closed.

Compile Listing 13-22 as follows:

javac FormatterDemo.java

Run this application as follows:

java FormatterDemo
You should observe the following output:

```
123
1237b
1237bX
1237bX0.100000
1237bX0.100000Hello, World
1237bX0.100000Hello, World 98.38
1237bX0.100000Hello, World 98.3800123
1237bX0.100000Hello, World 98.3800123 123
Exception in thread "main" java.util.MissingFormatArgumentException: Format specifier 'd'
at java.util.Formatter.format(Formatter.java:2487)
at java.util.Formatter.format(Formatter.java:2423)
at FormatterDemo.main(FormatterDemo.java:24)
```

The first thing to notice about the output is that each `format()` call appends formatted output to the previously formatted output. The second thing to notice is that `java.util.MissingFormatArgumentException` is thrown when you don't specify a needed argument.

Note: `MissingFormatArgumentException` is one of several formatter exception types. These exception types subtype the `java.util.IllegalArgumentException` type.

If you aren’t happy with this concatenated output, there are two ways to solve the problem:

- Instantiate a new `Formatter` instance, as in `formatter = new Formatter();`, before calling `format()`. This ensures that a new default and empty string builder is created.
- Create your own `StringBuilder` instance and pass it to a constructor such as `Formatter(Appendable a)`. After outputting the formatted content, invoke `StringBuilder`'s `void setLength(int newLength)` method with 0 as the argument to erase previous content.

It's cumbersome to have to create and manage a `Formatter` object when all you want to do is to achieve something equivalent to the C language’s `printf()` function. Java addresses this situation by adding formatter support to the `java.io.PrintStream` class.

Of the various formatter-oriented methods added to `PrintStream`, you’ll often invoke `PrintStream printf(String format, Object... args)`. After sending its formatted content to the print stream, this method returns a reference to this stream so that you can chain method calls together.

Listing 13-23 provides a small `printf()` demonstration.
Listing 13-23. Formatting via printf()

```java
public class FormatterDemo {
    public static void main(String[] args) {
        System.out.printf("%04X%n", 478);
        System.out.printf("Current date: %1$tb %1$te, %1$tY%n",
                System.currentTimeMillis());
    }
}
```

Listing 13-23's main() method invokes System.out.printf() twice. The first invocation formats 32-bit integer 478 into a four-digit hexadecimal string with a leading zero and uppercase hexadecimal digits. The second invocation formats the current millisecond value returned from System.currentTimeMillis() into a date. The tb conversion specifies an abbreviated month name (such as Jan), the te conversion specifies the day of the month (such as 1 through 31), and the tY conversion specifies the year (formatted with at least four digits, with leading 0s as necessary).

Compile Listing 13-23 (javac FormatterDemo.java), and run the application (java FormatterDemo). You should observe output similar to the output shown below:

```
01DE
Current date: Jan 14, 2014
```

Note  For more information on Formatter and its supported format specifiers, I refer you to Formatter's Java documentation. You might also want to check out the documentation on the Formattable interface and FormattableFlags class to learn about customizing Formatter.

Working with Scanner

Java 5 introduced the java.util.Scanner class to parse input characters into primitive types, strings, and big integers/big decimals with the help of regular expressions. Scanner declares several constructors for scanning content originating from diverse sources. For example, Scanner(InputStream source) creates a scanner for scanning the specified input stream, whereas Scanner(String source) creates a scanner for scanning the specified string.

A Scanner instance uses a delimiter pattern, which matches whitespace by default, to break its input into discrete values. After creating this instance, you can call one of the “hasNext” methods to verify that an anticipated character sequence is present for scanning. For example, you could call boolean hasNextDouble() to determine whether or not the next sequence of characters can be scanned into a double precision floating-point value.

When the value is present, you would call the appropriate “next” method to scan the value. For example, you would call double nextDouble() to scan this sequence and return a double containing its value.
When you’re finished with the scanner, invoke its `void close()` method. Beyond closing the scanner, this method also closes the underlying input source when this source’s class implements the `Closeable` interface. If the scanner has been closed, this method has no effect. Any attempt to scan after calling this method will result in `IllegalStateException`.

The following example shows you how to create a scanner for scanning values from standard input and then scanning an integer followed by a double precision floating-point value:

```java
Scanner sc = new Scanner(System.in);
if (sc.hasNextInt())
  i = sc.nextInt();
if (sc.hasNextDouble())
  d = sc.nextDouble();
```

Listing 13-24 presents a more realistic (menu-oriented) example.

**Listing 13-24. Scanning Input in a Menu Context**

```java
import java.util.Scanner;

public class ScannerDemo
{
  public static void main(String[] args)
  {
    Scanner scanner = new Scanner(System.in);
    while (true)
    {
      System.out.printf("%nMenu Options%n%n");
      System.out.println("1: Frequency Count");
      System.out.printf("2: Quit%n%n");
      System.out.print("Enter your selection (1 or 2): ");
      int selection = scanner.nextInt();
      scanner.nextLine();
      if (selection == 1)
      {
        System.out.printf("%nEnter sentence: ");
        String sentence = scanner.nextLine();
        System.out.print("Enter index: ");
        int index = scanner.nextInt();
        int count = 0;
        for (int i = 0; i < sentence.length(); i++)
        {
          if (sentence.charAt(i) == sentence.charAt(index))
            count++;
        }
        System.out.printf("Count of [%c] in [%s]: %d%n", sentence.charAt(index), sentence, count);
      }
      else
      {
        if (selection == 2)
          break;
      }
    }
    scanner.close();
  }
}
```
Listing 13-24’s main() method creates a scanner that scans input from the standard input stream and then enters a while loop. Each of the loop iterations presents a two-option menu and prompts the user to select one of these options.

Option selection is made via a scanner.nextInt() method call. Because nextInt() doesn’t consume the line terminator following the selection number, a call to Scanner’s void nextLine() method is made to skip over the line terminator so as not to affect sentence entry (when option 1 is chosen).

If the user selected option 1, the user is prompted to enter a sentence along with the zero-based index of one of the sentence characters. The sentence is then iterated over, and all occurrences of the indexed character are tallied. This count is subsequently output.

If the user selected option 2, the loop is broken and the application ends.

Compile Listing 13-24 as follows:

```
javac ScannerDemo.java
```

Run this application as follows:

```
javac ScannerDemo
```

The following output reveals one run of this application:

```
Menu Options
1: Frequency Count
2: Quit

Enter your selection (1 or 2): 1

Enter sentence: This is a test.
Enter index: 2
Count of [i] in [This is a test.]: 2

Menu Options
1: Frequency Count
2: Quit

Enter your selection (1 or 2): 2
```

**Note** To learn more about Scanner, check out this class’s Java documentation.
EXERCISES

The following exercises are designed to test your understanding of Chapter 13’s content.

1. Define New I/O.
2. What is a buffer?
3. Identify a buffer’s four properties.
4. What happens when you invoke Buffer’s array() method on a buffer backed by a read-only array?
5. What happens when you invoke Buffer’s flip() method on a buffer?
6. What happens when you invoke Buffer’s reset() method on a buffer where a mark has not been set?
7. True or False: Buffers are thread-safe.
8. Identify the classes that extend the abstract Buffer class.
9. How do you create a byte buffer?
10. Define view buffer.
11. How is a view buffer created?
12. How do you create a read-only view buffer?
13. Identify ByteBuffer’s methods for storing a single byte in a byte buffer and fetching a single byte from a byte buffer.
14. What causes BufferOverflowException or BufferUnderflowException to occur?
15. What is the equivalent of executing buffer.flip();?
16. True or false: Calling flip() twice returns you to the original state.
17. What is the difference between Buffer’s clear() and reset() methods?
18. What does ByteBuffer’s compact() method accomplish?
19. What is the purpose of the ByteOrder class?
20. Define direct byte buffer.
21. How do you obtain a direct byte buffer?
22. What is a channel?
23. What capabilities does the Channel interface provide?
24. Identify the three interfaces that directly extend Channel.
25. True or false: A channel that implements InterruptibleChannel is asynchronously closeable.
26. Identify the two ways to obtain a channel.
27. Define scatter/gather I/O.
28. What interfaces are provided for achieving scatter/gather I/O?
29. Define file channel.
30. True or false: File channels don’t support scatter/gather I/O.
31. Define exclusive lock and shared lock.
32. What is the fundamental difference between FileChannel’s lock() and tryLock() methods?
33. What does the FileLock lock() method do when either a lock is already held that overlaps this lock request or another thread is waiting to acquire a lock that will overlap with this request?
34. Specify the pattern that you should adopt to ensure that an acquired file lock is always released.
35. What method does FileChannel provide for mapping a region of a file into memory?
36. Identify the three file-mapping modes.
37. Which file-mapping mode corresponds to copy-on-write?
38. Identify the FileChannel methods that optimize the common practice of performing bulk transfers.
39. True or false: Socket channels are selectable and can function in nonblocking mode.
40. Identify the three classes that describe socket channels.
41. True or false: Datagram channels are not thread-safe.
42. Why do socket channels support nonblocking mode?
43. How would you obtain a socket channel’s associated socket?
44. How do you obtain a server socket channel?
45. Define selector.
46. Identify the three main types that support selectors.
47. True or false: File channels can be used with selectors.
48. Define regular expression.
49. What does the Pattern class accomplish?
50. What do Pattern’s compile() methods do when they discover illegal syntax in their regular expression arguments?
51. What does the Matcher class accomplish?
52. What is the difference between Matcher’s matches() and lookingAt() methods?
53. Define character class.
54. Identify the various kinds of character classes.
55. Define capturing group.
56. What is a zero-length match?
57. Define quantifier.
58. What is the difference between a greedy quantifier and a reluctant quantifier?
59. How do possessive and greedy quantifiers differ?
60. Identify the two main classes that contribute to the NIO printf-style formatting facility.

61. What does the %n format specifier accomplish?

62. Refactor Listing 11–11 (Chapter 11’s Copy application) to use the ByteBuffer and FileChannel classes in partnership with FileInputStream and FileOutputStream.

63. Create a ReplaceText application that takes input text, a pattern that specifies text to replace, and replacement text command-line arguments, and uses Matcher’s String replaceAll(String replacement) method to replace all matches of the pattern with the replacement text (passed to replacement). For example, java ReplaceText "too many embedded spaces" "\s+" would output too many embedded spaces with only a single space character between successive words.

64. Create a ValidateInput application that reads the contents of standard input (via the Scanner class) and views this content as a sequence of lines with each line containing a string-based name followed by an integer-based age. For example, one line might contain Jack 40 and another line might contain Jill 32. If the current line is empty, the application should output name field missing. If the current line contains a name field only, the application should output age field missing. After reading a line, the application should output the line followed by a 1-based line number. It should extract the individual name and age and output, for example, Name: Jack and Age: 40. Include the current line number when outputting the line and individual messages. In addition to the previous lines, test this application with a line containing George only, a line containing 42 only, and an empty line.

Summary

Java 1.4 introduced a more powerful I/O architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture largely consists of buffers, channels, selectors, regular expressions, and charsets, and it is commonly known as New I/O (NIO).

NIO is based on buffers. A buffer is an object that stores a fixed amount of data to be sent to or received from an I/O service (a means for performing input/output). It sits between an application and a channel that writes the buffered data to the service or reads the data from the service and deposits it into the buffer. Java supports buffers by providing the Buffer class, assorted subclasses, and the ByteOrder typesafe enumeration.

Channels partner with buffers to achieve high-performance I/O. A channel is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that’s capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations. Java supports channels by providing the Channel interface and related types.

Selectors let you achieve readiness selection in a Java context. Readiness selection offloads to the operating system the work involved in checking for I/O stream readiness to perform write, read, and other operations. The operating system is instructed to observe a group of channels and return some indication of which channels are ready to perform a specific operation (such as read) or operations (such as accept and read). This capability lets a thread multiplex a potentially huge number of active channels by using the readiness information provided by the operating system. In this way, network
servers can handle large numbers of network connections; they are vastly scalable. Java supports selectors by offering the SelectableChannel, SelectionKey, and Selector classes.

Text-processing applications often need to match text against patterns (character strings that concisely describe sets of strings that are considered to be matches). For example, an application might need to locate all occurrences of a specific word pattern in a text file so that it can replace those occurrences with another word. NIO includes regular expressions to help text-processing applications perform pattern matching with high performance. Java supports regular expressions by providing the Pattern and Matcher classes.

Charsets combine coded character sets with character-encoding schemes. They’re used to translate between byte sequences and the characters that are encoded into these sequences. Java supports charsets by providing CharSet and related classes.

The description for JSR 51 (the NIO JSR) indicates that a simple printf-style formatting facility was proposed for inclusion in NIO. This facility consists of formatted output and formatted input. Because the C language versions of these features depend on varargs, and because support for varargs wasn’t added to Java until Java 5, this facility didn’t debut in Java 1.4. Instead, the printf facility was deferred to Java 5, which added Formatter and related types to perform formatted output and Scanner, which doesn’t depend on varargs, to perform formatted input.

Chapter 14 focuses on database access. You first encounter the Java DB and SQLite database products and then learn how to use the JDBC API to create/access their databases.
Accessing Databases

Applications often need to access databases to store and retrieve various kinds of data. A *database* ([http://en.wikipedia.org/wiki/Database](http://en.wikipedia.org/wiki/Database)) is an organized collection of data. Although there are many kinds of databases (such as hierarchical, object-oriented, and relational), *relational databases*, which organize data into tables that can be related to each other, are common.

**Note** In a relational database, each row stores a single item (such as an employee) and each column stores a single item attribute (such as an employee’s name).

Except for the most trivial of databases (such as Chapter 11’s flat file database based on a single data file), databases are created and managed through a *database management system (DBMS)*—see [http://en.wikipedia.org/wiki/Database_management_system](http://en.wikipedia.org/wiki/Database_management_system). Relational DBMSs (RDBMSs) support *Structured Query Language (SQL)* for working with tables and more.

**Note** For brevity, I assume that you’re familiar with SQL. If not, you might want to check out Wikipedia’s “SQL” entry ([http://en.wikipedia.org/wiki/SQL](http://en.wikipedia.org/wiki/SQL)) for an introduction.

Java supports database access and creation (and more) via its relational database-oriented JDBC (Java Database Connectivity) API. Because you need an RDBMS before you can explore JDBC, this chapter first introduces you to Java DB, which is included with the JDK, followed by the popular SQLite ([http://en.wikipedia.org/wiki/Sqlite](http://en.wikipedia.org/wiki/Sqlite)). This chapter then focuses on JDBC.
Android offers an alternative to JDBC via its android.database and android.database.sqlite packages, which are the preferred means for accessing databases from an Android application. Although Android supports JDBC by including this API and an undocumented JDBC driver (I discuss JDBC drivers later in this chapter), you should focus on using Android’s database access alternative when developing an Android application that requires database access. Because you still might find JDBC useful, especially when creating a non-Android application, I present JDBC in this chapter.

Introducing Java DB

First introduced by Sun Microsystems as part of JDK 6 (and not included in the JRE) to give developers an RDBMS to test their JDBC code, Java DB is a distribution of Apache’s open-source Derby product, which is based on IBM’s Cloudscape RDBMS code base. This pure-Java RDBMS is also bundled with JDK 7 (but not in the JRE). It’s secure, supports JDBC and SQL (including transactions, stored procedures, and concurrency), and has a small footprint—its core engine and JDBC driver occupy approximately 2MB.

Java DB is capable of running in an embedded environment or in a client/server environment. In an embedded environment, where an application accesses the database engine via Java DB’s embedded driver, the database engine runs in the same virtual machine as the application. Figure 14-1 illustrates the embedded environment architecture, where the database engine is embedded in the application.

![Figure 14-1. No separate processes are required to start up or shut down an embedded database engine](image-url)
In a client/server environment, client applications and the database engine run in separate virtual machines. A client application accesses the network server through Java DB’s client driver. The network server, which runs in the same virtual machine as the database engine, accesses the database engine through the embedded driver. Figure 14-2 illustrates this architecture.

**Figure 14-2. Multiple clients communicate with the same database engine through the network server**

Java DB implements the database portion of the architectures shown in Figures 14-1 and 14-2 as a directory with the same name as the database. Within this directory, Java DB creates a log directory to store transaction logs, a seg0 directory to store the data files, and a service.properties file to store configuration parameters.

**Note** Java DB doesn’t provide an SQL command to drop (destroy) a database. Destroying a database requires that you manually delete its directory structure.
Java DB Installation and Configuration

When you install JDK 7 with the default settings, the bundled Java DB is installed into %JAVA_HOME%\db on Windows platforms, or into the db subdirectory in the equivalent location on Unix/Linux platforms. (For convenience, I adopt the Windows convention when presenting environment variable paths.)

Note I focus on Java DB 10.8.2.2 in this chapter because it’s included with JDK 7 build 1.7.0_06-b24, which is the Java build on which this book is based.

The db directory contains five files and the following pair of subdirectories:

- The bin directory contains scripts for setting up embedded and client/server environments, running command-line tools, and starting/stopping the network server. You should add this directory to your PATH environment variable so that you can conveniently execute its scripts from anywhere in the filesystem.

- The lib directory contains various JAR files that house the engine library (derby.jar), the command-line tools libraries (derbytools.jar and derbyrun.jar), the network server library (derbynet.jar), the network client library (derbyclient.jar), and various locale-specific libraries. This directory also contains derby.war, which is used to register the network servlet (see http://en.wikipedia.org/wiki/Java_Servlet) at the /derbynet relative path—it’s also possible to manage the Java DB network server remotely via the servlet interface (see http://db.apache.org/derby/docs/10.8/adminguide/cadminservlet98430.html).

Before you can run the tools and start/stop the network server, you must set the DERBY_HOME environment variable. Set this variable for Windows via set DERBY_HOME=%JAVA_HOME%\db, and for Unix (Korn shell) via export DERBY_HOME=$JAVA_HOME/db. (This setting will not persist past the current command shell session unless you make it permanent.)

Note The embedded and client/server environment setup scripts refer to a DERBY_INSTALL environment variable. According to the “Re: DERBY_INSTALL and DERBY_HOME” mail item (www.mail-archive.com/derby-dev@db.apache.org/msg22098.html), DERBY_HOME is equivalent to and replaces DERBY_INSTALL for consistency with other Apache projects.
You must also set the CLASSPATH environment variable. The easiest way to set this environment variable is to run a script file included with Java DB. Windows and Unix/Linux versions of various “setxxxCP” script files (which extend the current classpath) are located in the %JAVA_HOME%\db\bin directory. The script file(s) to run will depend on whether you work with the embedded or client/server environment:

- For the embedded environment, invoke setEmbeddedCP to add derby.jar and derbytools.jar to the classpath.
- For the client/server environment, invoke setNetworkServerCP to add derbynet.jar and derbytools.jar to the classpath. In a separate command window, invoke setNetworkClientCP to add derbyclient.jar and derbytools.jar to the classpath.

**Java DB Demos**

For JDK 7, the Java DB demos are included with other Java demos in a separate distribution file—see [www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html](http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html). After downloading and unarchiving the distribution file, you can move it to %JAVA_HOME%. If you’re running Windows 7, you’ll also need to ensure that Java DB can write files to subdirectories of C:\Program Files (64-bit JDK) or C:\Program Files (x86) (32-bit JDK). Otherwise, you’ll encounter “access denied” errors.

The %JAVA_HOME%\demo\db\programs directory contains HTML documentation that describes the demos included with Java DB; the demo.html file is the entry point into this documentation. These demos include a simple JDBC application for working with Java DB, a network server sample program, and sample programs that are introduced in the *Working with Derby* manual.

Note: The *Working with Derby* manual underscores Java DB’s Derby heritage. You can download this manual and other Derby manuals from the documentation section ([http://db.apache.org/derby/manuals/index.html](http://db.apache.org/derby/manuals/index.html)) of Apache’s Derby project site ([http://db.apache.org/derby/index.html](http://db.apache.org/derby/index.html)).

For brevity, I’ll focus only on the simple JDBC application that’s located in the programs directory’s simple subdirectory. This application runs in either the default embedded environment or the client/server environment. It creates and connects to a derbyDB database, introduces a table into this database, performs SQL insert/update/select operations on this table, drops (removes) the table, and disconnects from the database.

To run this application in the embedded environment, open a command window and make sure that the DERBY_HOME and CLASSPATH environment variables have been set properly; then invoke setEmbeddedCP to set the classpath. Assuming that simple is the current directory, invoke java SimpleApp or java SimpleApp embedded to run this application. You should observe the following output:

```
SimpleApp starting in embedded mode
Loaded the appropriate driver
Connected to and created database derbyDB
```
CHAPTER 14: Accessing Databases

Created table location
Inserted 1956 Webster
Inserted 1910 Union
Updated 1956 Webster to 180 Grand
Updated 180 Grand to 300 Lakeshore
Verified the rows
Dropped table location
Committed the transaction
Derby shut down normally
SimpleApp finished

This output reveals that an application running in the embedded environment shuts down the database engine before exiting. This is done to perform a checkpoint and release resources. When this shutdown doesn't occur, Java DB notes the absence of the checkpoint, assumes a crash, and causes recovery code to run before the next database connection (which takes longer to complete).

Tip When running SimpleApp (or any other Java DB application) in the embedded environment, you can determine where the database directory will be created by setting the derby.system.home property. For example, java -Dderby.system.home=c:\temp SimpleApp causes derbyDB to be created in the temp subdirectory of the root directory of the C: drive on the Windows 7 platform.

To run this application in the client/server environment, you need to start the network server and run the application in separate command windows.

In one command window, set DERBY_HOME. Start the network server via the startNetworkServer script (located in %JAVA_HOME%/db/bin), which takes care of setting the classpath. You should see output similar to this:

Fri Oct 18 13:21:36 CST 2013 : Apache Derby Network Server - 10.8.2.2 - (1181258) started and ready to accept connections on port 1527

In the other command window, set DERBY_HOME followed by CLASSPATH (via setNetworkClientCP). Assuming that the simple directory is current, execute the java SimpleApp derbyClient command line to run this application. This time, you should observe the following output:

SimpleApp starting in derbyclient mode
Loaded the appropriate driver
Connected to and created database derbyDB
Created table location
Inserted 1956 Webster
Inserted 1910 Union
Updated 1956 Webster to 180 Grand
Updated 180 Grand to 300 Lakeshore
Verified the rows
CHAPTER 14: Accessing Databases

Dropped table location
Committed the transaction
SimpleApp finished

Notice that the database engine is not shut down in the client/server environment. Although not indicated in the output, there's a second difference between running SimpleApp in the embedded and client/server environments. In the embedded environment, the derbyDB database directory is created in the simple directory. In the client/server environment, this database directory is created in the directory that was current when you executed startNetworkServer.

When you're finished playing with SimpleApp in the client/server environment, you should shut down the network server and database engine. Accomplish this task by invoking the stopNetworkServer script (located in %JAVA_HOME%\db\bin). You can also shut down (or start and otherwise control) the network server by running the NetworkServerControl script (also located in %JAVA_HOME%\db\bin). For example, NetworkServerControl shutdown shuts down the network server and database engine.

Java DB Command-Line Tools

The %JAVA_HOME%\db\bin directory contains sysinfo, ij, and dblook Windows and Unix/Linux script files for launching command-line tools:

- Run sysinfo to view the Java environment/Java DB configuration.
- Run ij to run scripts that execute ad hoc SQL commands and perform repetitive tasks.
- Run dblook to view all or part of a database’s Data Definition Language (DDL).

If you experience trouble with Java DB (such as not being able to connect to a database), you can run sysinfo to find out if the problem is configuration-related. This tool reports various settings under the Java Information, Derby Information, and Locale Information headings. It outputs the following information on my platform:

```
------------------ Java Information ------------------
Java Version:    1.7.0_06
Java Vendor:     Oracle Corporation
Java home:       C:\Program Files\Java\jdk1.7.0_06\jre
Java classpath:  C:\PROGRA~1\Java\JDK17~2.0_0\db\lib\derbyclient.jar;C:\PROGRA~1\Java\JDK17~2.0_0\db\lib\derbytools.jar;.;C:\Program Files (x86)\QuickTime\QTSystem\QTJava.zip;C:\PROGRA~1\Java\JDK17~2.0_0\db\lib\derbynet.jar;C:\PROGRA~1\Java\JDK17~2.0_0\db\lib\derbyclient.jar;C:\PROGRA~1\Java\JDK17~2.0_0\db\lib\derbytools.jar
OS name:         Windows 7
OS architecture: amd64
OS version:      6.1
Java user name:  Owner
Java user home:  C:\Users\Owner
Java user dir:   C:\PROGRA~1\Java\jdk1.7.0_06\db\bin
java.specification.name: Java Platform API Specification
java.specification.version: 1.7
java.runtime.version: 1.7.0_06-b24
```
The `ij` script is useful for creating a database and initializing a user's schema (a namespace that logically organizes tables and other database objects) by running a script file that specifies the appropriate DDL statements. For example, you've created an EMPLOYEES table with its NAME and PHOTO columns and you have created a `create_emp_schema.sql` script file in the current directory that contains the following line:

```sql
CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB);
```

The following embedded `ij` script session creates the employees database and EMPLOYEES table:

```bash
C:\db>ij
ij version 10.8
ij> connect 'jdbc:derby:employees;create=true';
ij> run 'create_emp_schema.sql';
```
ij> CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB);
0 rows inserted/updated/deleted
ij> disconnect;
ij> exit;
C:>\db>

The connect command causes the employees database to be created—I'll have more to say about this command's syntax when I introduce JDBC later in this chapter. The run command causes create_emp_schema.sql to execute, and the subsequent pair of lines is generated as a result.

The CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB); line is an SQL statement for creating a table named EMPLOYEES with NAME and PHOTO columns. Data items entered into the NAME column are of SQL type VARCHAR (a varying number of characters—a string) with a maximum of 30 characters, and data items entered into the PHOTO column are of SQL type BLOB (a binary large object, such as an image).

**Note** I specify SQL statements in uppercase, but you can also specify them in lowercase or mixed case.

After run 'create_emp_schema.sql' finishes, the specified EMPLOYEES table is added to the newly created employees database. To verify the table's existence, run dblook against the employees directory, as the following session demonstrates:

C:\db>dblook -d jdbc:derby:employees
-- Timestamp: 2012-11-25 16:13:42.693
-- Source database is: employees
-- Connection URL is: jdbc:derby:employees
-- appendLogs: false

-- ----------------------------------------------
-- DDL Statements for tables
-- ----------------------------------------------

CREATE TABLE "APP"."EMPLOYEES" ("NAME" VARCHAR(30), "PHOTO" BLOB(2147483647));

C:\db>

All database objects (such as tables and indexes) are assigned to user and system schemas, which logically organize these objects in the same way that packages logically organize classes. When a user creates or accesses a database, Java DB uses the specified username as the namespace name for newly added database objects. In the absence of a username, Java DB chooses APP, as the preceding session output shows.
Introducing SQLite

SQLite (http://sqlite.org/) is a very simple and popular RDBMS. Basically, it implements a self-contained, serverless, zero-configuration, transactional SQL database engine; and is considered to be the most widely deployed database engine in the world. For example, SQLite is found in Mozilla Firefox, Google Chrome, and other web browsers. It’s also found in Google Android, Apple iOS, and other mobile operating systems.

Note To learn what sets SQLite apart from other RDBMs, visit the “Distinctive Features of SQLite” page at http://sqlite.org/different.html. As well as learning about features such as the aforementioned zero-configuration, you’ll learn about features such as manifest typing, in which you can store any value of any data type in any column regardless of the declared type of that column.

To introduce yourself to SQLite, visit the SQLite home page at http://sqlite.org/. You can explore online documentation (http://sqlite.org/docs.html), download SQLite software (http://sqlite.org/download.html), and so on. Regarding downloads, you can download source code, documentation, and precompiled binaries for the Linux, Mac OS X (x86), and Windows platforms.

I downloaded the sqlite-shell-win32-x86-3071401.zip distribution file for my Windows 7 platform. This archive contains a single sqlite3 executable, which offers a command-line shell for accessing and modifying SQLite databases. According to the SQLite downloads page, this program is compatible with all versions of SQLite through version 3.7.14.1 (and beyond). (The Android SDK for Windows also includes sqlite3.exe but not necessarily the same version.)

You can specify sqlite3 with a database filename argument (such as sqlite3 employees) to create the database file (employees, for example) when it doesn’t exist (you must create a table at least) or open the existing file and enter this tool’s shell from where you can execute sqlite3-specific, dot-prefixed commands and SQL statements. As Figure 14-3 shows, you can also specify sqlite3 without an argument and enter the shell.
CHAPTER 14: Accessing Databases

Figure 14-3 reveals the prologue that greets you after entering the sqlite3 shell, which is indicated by the sqlite> prompt from where you enter commands. It also reveals part of the help text that's presented when you type the sqlite3-specific .help command.

**Tip** You can create a database after specifying sqlite3 without an argument by entering the appropriate SQL statements to create and populate desired tables (and possibly create indexes) and then invoking .backup filename (where filename identifies the file that stores the database) before exiting sqlite3.

While discussing Java DB command-line tools, I presented a small employee-oriented database example consisting of an employees database and a create_emp_schema.sql script file that contains the following SQL statement for creating an EMPLOYEES table (consisting of names and photos).

```
CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB);
```

Let's find out how to create this database and table with sqlite3.

At the command line, execute sqlite3 employees. At the resulting sqlite> command prompt, execute the aforementioned SQL statement, and then execute .quit to quit sqlite3. You should now observe an employees file in the same directory as sqlite3.
Re-execute sqlite3 employees. At the sqlite> command prompt, execute .tables. You should observe a single output line consisting of EMPLOYEES. Now execute .schema employees (case isn’t significant) and you should see the aforementioned CREATE TABLE statement.

You can continue to play with sqlite3 and the employees database/EMPLOYEES table. For example, you could insert a single row of data into the EMPLOYEES table via the following INSERT statement and then select/output this row via the following SELECT statement:

```
INSERT INTO EMPLOYEES VALUES('Duke', null);
SELECT * FROM EMPLOYEES;
```

You should observe the following line as the result—nothing appears for the photo because of its null value:

```
DUKE|
```

## Accessing Databases via JDBC

JDBC is an API (associated with the java.sql and javax.sql packages—I mainly focus on java.sql in this chapter) for communicating with RDBMSs in an RDBMS-independent manner. You can use JDBC to perform various database operations, such as submitting SQL statements that tell the RDBMS to create a table and to update or query tabular data.

### Data Sources, Drivers, and Connections

Although JDBC is typically used to communicate with RDBMSs, it also can be used to communicate with a flat file database. For this reason, JDBC uses the term data source (a data-storage facility ranging from a simple file to a complex relational database managed by an RDBMS) to abstract the source of data.

Because data sources are accessed in different ways (such as Chapter 11’s flat file database is accessed via methods of the java.io.RandomAccessFile class, whereas Java DB and SQLite databases are accessed via SQL statements), JDBC uses drivers (classfile plug-ins) to abstract over their implementations. This abstraction lets you write an application that can be adapted to an arbitrary data source without having to change a single line of code (in most cases). Drivers are implementations of the java.sql.Driver interface.

JDBC recognizes four types of drivers.

- **Type 1 drivers** implement JDBC as a mapping to another data-access API (such as Open Database Connectivity, or ODBC—see [http://en.wikipedia.org/wiki/ODBC](http://en.wikipedia.org/wiki/ODBC)). The driver converts JDBC method calls into function calls on the other library. The JDBC-ODBC Bridge Driver is an example and isn’t supported by Oracle. It was commonly used in the early days of JDBC when other kinds of drivers were uncommon.

- **Type 2 drivers** are written partly in Java and partly in native code. They interact with a data source-specific native client library and are not portable for this reason. Oracle’s OCI (Oracle Call Interface) client-side driver is an example.
Type 3 drivers don’t depend on native code and communicate with a middleware server (a server that sits between the application client and the data source) via an RDBMS-independent protocol. The middleware server then communicates the client’s requests to the data source.

Type 4 drivers don’t depend on native code and implement the network protocol for a specific data source. The client connects directly to the data source instead of going through a middleware server.

Before you can communicate with a data source, you need to establish a connection. JDBC provides the java.sql.DriverManager class and the javax.sql.DataSource interface for this purpose.

DriverManager lets an application connect to a data source by specifying a URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.x drivers located via the classpath. (Pre-JDBC 4.x drivers must be loaded manually.)

DataSource hides connection details from the application to promote data source portability and is preferred over DriverManager for this reason. Because a discussion of DataSource is somewhat involved (and is typically used in a Java EE context), I focus on DriverManager in this chapter.

Before letting you obtain a data source connection, early JDBC versions required you to explicitly load a suitable driver by specifying Class.forName() with the name of the class that implements the Driver interface. For example, the JDBC-ODBC Bridge driver was loaded via Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); Later JDBC versions relaxed this requirement by letting you specify a list of drivers to load via the jdbc.drivers system property. DriverManager would attempt to load all of these drivers during its initialization.

Under Java 7, DriverManager first loads all drivers identified by the jdbc.drivers system property. It then uses the java.util.ServiceLoader-based service provider mechanism to load all drivers from accessible driver JAR files so that you don’t have to explicitly load drivers. This mechanism requires a driver to be packaged into a JAR file that includes META-INF/services/java.sql.Driver. The java.sql.Driver text file must contain a single line that names the driver’s implementation of the Driver interface.

Each loaded driver instantiates and registers itself with DriverManager via DriverManager’s void registerDriver(Driver driver) class method. When invoked, a getConnection() method walks through registered drivers, returning an implementation of the java.sql.Connection interface from the first driver that recognizes getConnection()’s JDBC URL. (You might want to check out DriverManager’s source code to see how this is done.)

Note To maintain data source-independence, much of JDBC consists of interfaces. Each driver provides implementations of the various interfaces.

To connect to a data source and obtain a Connection instance, call one of DriverManager’s Connection getConnection(String url), Connection getConnection(String url, Properties info), or Connection getConnection(String url, String user, String password) methods. With
either method, the url argument specifies a string-based URL that starts with the jdbc: prefix and continues with data source-specific syntax.

Consider Java DB. The URL syntax varies depending on the driver. For the embedded driver (when you want to access a local database), this syntax is as follows:

```
jdbc:derby:databaseName;URLAttributes
```

For the client driver (when you want to access a remote database, although you can also access a local database with this driver), this syntax is as follows:

```
jdbc:derby://host:port/databaseName;URLAttributes
```

With either syntax, URLAttributes is an optional sequence of semicolon-delimited name=value pairs. For example, create=true tells Java DB to create a new database.

The following example demonstrates the first syntax by telling JDBC to load the Java DB embedded driver and create the database named testdb on the local host:

```
Connection con = DriverManager.getConnection("jdbc:derby:testdb;create=true");
```

The following example demonstrates the second syntax by telling JDBC to load the Java DB client driver and create the database named testdb on port 8500 of the xyz host:

```
Connection con;
con = DriverManager.getConnection("jdbc:derby://xyz:8500/testdb;create=true");
```

Consider SQLite. The Xerial project ([www.xerial.org/trac/Xerial](http://www.xerial.org/trac/Xerial)) provides the SQLite JDBC driver ([www.xerial.org/trac/Xerial/wiki/SQLiteJDBC](http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC)) for testing JDBC with SQLite. Point your browser to [https://bitbucket.org/xerial/sqlite-jdbc/downloads](https://bitbucket.org/xerial/sqlite-jdbc/downloads) and download an appropriate driver JAR file (such as sqlite-jdbc-3.7.2.jar).

For creating an actual file in which to store the database, the URL syntax for the Xerial SQLite driver is as follows:

```
jdbc:sqlite:databaseName
```

The following examples demonstrate this syntax for connecting to a database file (which is created when it doesn’t exist) named sample.db:

```
Connection con1 = DriverManager.getConnection("jdbc:sqlite:sample.db");
Connection con2 = DriverManager.getConnection("jdbc:sqlite:C:/temp/sample.db");
```

The first example obtains a connection to the current directory’s sample.db file; the second example obtains a connection to a sample.db file in the C:\temp directory.

SQLite also supports in-memory database management, which doesn’t create any database files. The following example shows you how to connect to an existing in-memory database:

```
Connection con = DriverManager.getConnection("jdbc:sqlite::memory:");
```
CHAPTER 14: Accessing Databases

The following example shows you how to create and obtain a connection to an in-memory database:

```java
Connection con = DriverManager.getConnection("jdbc:sqlite:);
```

**Note** For the most part, this chapter's applications can be used with either the Java DB embedded driver connection syntax or the non-in-memory SQLite driver connection syntax.

## Exceptions

`DriverManager`'s `getConnection()` methods (and other JDBC methods in the various JDBC interfaces) throw `java.sql.SQLException` or one of its subclasses when something goes wrong. Along with the methods it inherits from `java.lang.Throwable` (such as `String getMessage()`), `SQLException` declares various constructors (not discussed for brevity) and the following methods:

- `int getErrorCode()` returns a vendor-specific integer error code. Normally this value will be the actual error code returned by the underlying data source.
- `SQLException getNextException()` returns the `SQLException` instance chained to this `SQLException` object (via a call to `setNextException(SQLException ex)`) or null when there isn't a chained exception.
- `Iterator<Throwable> iterator()` returns an iterator over the chained `SQLExceptions` and their causes in proper order. The iterator will be used to iterate over each `SQLException` and its underlying cause (if any). You would normally not call this method but would instead use the enhanced for statement (discussed in Chapter 9), which calls `iterator()` when you need to iterate over the chain of `SQLExceptions`. (The Android documentation at the current time of writing reports this method to be obsolete.)
- `void setNextException(SQLException sqlex)` appends `sqlex` to the end of the chain. (The Android documentation at the current time of writing reports this method to be obsolete.)

One or more `SQLExceptions` might occur while processing a request, and the code that throws these exceptions can add them to a *chain* of `SQLExceptions` by invoking `setNextException()`. Also, an `SQLException` instance might be thrown as a result of a different exception (`java.io.IOException`, for example) which is known as that exception's *cause* (see Chapter 5).

SQL state error codes are defined by the ISO/ANSI and Open Group (X/Open) SQL standards. The error code is a five-character string consisting of a two-character class value followed by a three-character subclass value. Class value “00” indicates success, class value “01” indicates a warning, and other class values normally indicate an exception. Examples of SQL state error codes are 00000 (success) and 08001 (unable to connect to the data source).
Listing 14-1 presents a framework for structuring a JDBC application that connects to a JDBC or SQLite data source, performs some work, and responds to a thrown SQLException instance.

Listing 14-1. Architecting a Basic JDBC Application

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCDemo {
    final static String URL1 = "jdbc:derby:employee;create=true";
    final static String URL2 = "jdbc:sqlite:employee";

    public static void main(String[] args) {
        String url = null;
        if (args.length != 1) {
            System.err.println("usage 1: java JDBCDemo javadb");
            System.err.println("usage 2: java JDBCDemo sqlite");
            return;
        }
        if (args[0].equals("javadb"))
            url = URL1;
        else if (args[0].equals("sqlite"))
            url = URL2;
        else {
            System.err.println("invalid command-line argument");
            return;
        }
        Connection con = null;
        try {
            if (args[0].equals("sqlite"))
                Class.forName("org.sqlite.JDBC");
            con = DriverManager.getConnection(url);
            // Perform useful work. The following throw statement simulates a
            // JDBC method throwing SQLException.
            throw new SQLException("Unable to access database table",
                    new java.io.IOException("File I/O problem"));
        }
        catch (ClassNotFoundException cnfe) {
            System.err.println("unable to load sqlite driver");
        }
    }
}
```
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
}
finally
{
    if (con != null)
        try
        {
            con.close();
        }
        catch (SQLException sqle)
        {
            sqle.printStackTrace();
        }
}

Listing 14-1 requires that you run this application with the javadb or sqlite command-line argument. This argument determines which JDBC driver to use. If you specify sqlite as the argument, Xerial requires that the SQLite driver classfile be explicitly loaded, and this task is accomplished via the `Class.forName("org.sqlite.JDBC")` method call.

Next, a connection to the data source is obtained. When successful, IOException and SQLException objects are created, and the IOException instance is wrapped inside the SQLException instance (as its cause), which is subsequently thrown. The catch block that handles the SQL exception uses a while loop to demonstrate outputting the SQL exception and all chained exceptions.

Connections must be closed when no longer needed. Connection declares a void close() method for this purpose. This method is documented to throw SQLException.

Compile Listing 14-1 via the following command line:

```
javac JDBCDemo.java
```

Run this application via the following command line:

```
java JDBCDemo javadb
```
Assuming that Java DB hasn't been configured (by setting the DERBY_HOME and CLASSPATH environment variables), you should expect the following output:

```
SQL error : No suitable driver found for jdbc:derby:employee;create=true
SQL state : 08001
Error code: 0
Cause: null
```

Set the DERBY_HOME environment variable, and then execute setEmbeddedCP to install Java DB's embedded driver. Then re-execute `java JDBCDemo javadb`. This time, you should observe the following correct output:

```
SQL error : Unable to access database table
SQL state : null
Error code: 0
Cause: java.io.IOException: File I/O problem
```

Furthermore, an employee directory containing the database and a derby.log file should appear in the current directory.

Now, run this application via the following command:

```
java JDBCDemo sqlite
```

Assuming that SQLite hasn't been configured, you should observe the following output:

```
unable to load sqlite driver
```

This error message results from the thrown `java.lang.ClassNotFoundException` instance. This exception was thrown from the `Class.forName("org.sqlite.JDBC")` method call that attempted to load a nonexistent driver classfile.

You need to add the Xerial SQLite driver to the classpath when running `JDBCDemo`. Accomplish this task via the following command line:

```
java -cp sqlite-jdbc-3.7.2.jar;. JDBCDemo sqlite
```

Because of the previously created employee directory, you should observe the following output:

```
SQL error : [SQLITE_CANTOPEN] Unable to open the database file (out of memory)
SQL state : null
Error code: 0
Cause: null
```

Remove the employee directory (and derby.log for neatness) and re-execute the aforementioned command line. This time, you should observe the following correct output:

```
SQL error : Unable to access database table
SQL state : null
Error code: 0
Cause: java.io.IOException: File I/O problem
```
SQLException declares several subclasses (such as java.sql.BatchUpdateException—an error has occurred during a batch update operation). Many of these subclasses are categorized under java.sql.SQLNonTransientException- and java.sql.SQLTransientException-rooted class hierarchies in which SQLNonTransientException describes failed operations that cannot be retried without changing application source code or some aspect of the data source, and SQLTransientException describes failed operations that can be retried immediately.

**Statements**

After obtaining a connection to a data source, an application interacts with the data source by issuing SQL statements (such as CREATE TABLE, INSERT, SELECT, UPDATE, DELETE, and DROP TABLE). JDBC supports SQL statements via the java.sql.Statement, java.sql.PreparedStatement, and java.sql.CallableStatement interfaces. Furthermore, Connection declares various createStatement(), prepareStatement, and prepareCall() methods that return Statement, PreparedStatement, or CallableStatement implementation instances, respectively.

**Statement and ResultSet**

Statement is the easiest-to-use interface, and Connection’s Statement createStatement() method is the easiest-to-use method for obtaining a Statement instance. After calling this method, you can execute various SQL statements by invoking Statement methods such as the following:

- ResultSet executeQuery(String sql) executes a SELECT statement and (assuming no exception is thrown) provides access to its results via a java.sql.ResultSet instance.
- int executeUpdate(String sql) executes a CREATE TABLE, INSERT, UPDATE, DELETE, or DROP TABLE statement and (assuming no exception is thrown) typically returns the number of table rows affected by this statement.

I’ve created a second JDBCDemo application that demonstrates these methods. Listing 14-2 presents its source code.

**Listing 14-2. Creating, Inserting Values into, Querying, and Dropping an EMPLOYEES Table**

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo {
    final static String URL1 = "jdbc:derby:employee;create=true";
    final static String URL2 = "jdbc:sqlite:employee";

    public static void main(String[] args) {
        String url = null;
        if (args.length != 1)
```
{  
    System.err.println("usage 1: java JDBCDemo javadb");
    System.err.println("usage 2: java JDBCDemo sqlite");
    return;
}
if (args[0].equals("javadb"))
    url = URL1;
else
if (args[0].equals("sqlite"))
    url = URL2;
else
{
    System.err.println("invalid command-line argument");
    return;
}
Connection con = null;
try
{
if (args[0].equals("sqlite"))
    Class.forName("org.sqlite.JDBC");
con = DriverManager.getConnection(url);
Statement stmt = null;
try
{
    stmt = con.createStatement();
    String sql = "CREATE TABLE EMPLOYEES(ID INTEGER, NAME VARCHAR(30))";
    stmt.executeUpdate(sql);
    sql = "INSERT INTO EMPLOYEES VALUES(1, 'John Doe')";
    stmt.executeUpdate(sql);
    sql = "INSERT INTO EMPLOYEES VALUES(2, 'Sally Smith')";
    stmt.executeUpdate(sql);
    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEES");
    while (rs.next())
        System.out.println(rs.getInt("ID") + " " + rs.getString("NAME"));
    stmt.executeUpdate("DROP TABLE EMPLOYEES");
}
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
}
}
Listing 14-2 presents a similar architecture to Listing 14-1. For brevity, I won’t repeat the same instructions and examples that I presented while discussing SQL exceptions. Instead, I prefer to focus on new aspects of JDBC.

After successfully establishing a connection to the employee data source, main() creates a statement and uses it to execute SQL statements for creating, inserting values into, querying, and dropping an EMPLOYEES table.
The `executeQuery()` method returns a `ResultSet` object that provides access to a query’s tabular results. Each result set is associated with a cursor that provides access to a specific row of data. The cursor initially points before the first row; call `ResultSet`'s `boolean next()` method to advance the cursor to the next row. As long as there’s a next row, this method returns true; it returns false when there are no more rows to examine.

`ResultSet` also declares various methods for returning the current row’s column values based on their types. For example, `int getInt(String columnLabel)` returns the integer value corresponding to the `INTEGER`-based column identified by `columnLabel`. Similarly, `String getString(String columnLabel)` returns the string value corresponding to the `VARCHAR`-based column identified by `columnLabel`.

Tip If you don’t have column names but have one-based column indexes, call `ResultSet` methods such as `int getInt(int columnIndex)` and `String getString(int columnIndex)`. However, best practice is to call `int getInt(String columnLabel)`.

Compile Listing 14-2 and run this application as previously discussed—you will want to first delete the employee directory/file left behind by the previous application. You should observe the following output:

1 John Doe
2 Sally Smith

SQL’s `INTEGER` and `VARCHAR` types map to Java’s `int` and `java.lang.String` types. Table 14-1 presents a more complete list of type mappings.

### Table 14-1. SQL Type/Java Type Mappings

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td><code>java.sql.Array</code></td>
</tr>
<tr>
<td>BIGINT</td>
<td><code>Long</code></td>
</tr>
<tr>
<td>BINARY</td>
<td><code>byte[]</code></td>
</tr>
<tr>
<td>BIT</td>
<td><code>Boolean</code></td>
</tr>
<tr>
<td>BLOB</td>
<td><code>java.sql.Blob</code></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td><code>Boolean</code></td>
</tr>
<tr>
<td>CHAR</td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>CLOB</td>
<td><code>java.sql.Clob</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>java.sql.Date</code></td>
</tr>
<tr>
<td>DECIMAL</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>DOUBLE</td>
<td><code>Double</code></td>
</tr>
</tbody>
</table>

(continued)
CHAPTER 14: Accessing Databases

Check out http://docs.oracle.com/javase/1.5.0/docs/guide/jdbc/getstart/mapping.html for more information on type mappings.

### PreparedStatement

PreparedStatement is the next easiest-to-use interface, and Connection's PreparedStatement prepareStatement() method is the easiest-to-use method for obtaining a PreparedStatement instance—PreparedStatement is a subinterface of Statement.

Unlike a regular statement, a **preparedStatement** represents a precompiled SQL statement. The SQL statement is compiled to improve performance and prevent SQL injection (see http://en.wikipedia.org/wiki/SQL_injection), and the compiled result is stored in a PreparedStatement implementation instance.

You typically obtain this instance when you want to execute the same prepared statement multiple times. For example, you want to execute an SQL INSERT statement multiple times to populate a database table. Consider Listing 14-3.

#### Listing 14-3. Creating, Inserting Values via a Prepared Statement into, Querying, and Dropping an EMPLOYEES Table

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td>Double</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Int</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>REAL</td>
<td>Float</td>
</tr>
<tr>
<td>REF</td>
<td>java.sql.Ref</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Short</td>
</tr>
<tr>
<td>STRUCT</td>
<td>java.sql.Struct</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TINYINT</td>
<td>Byte</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

(Table 14-1. continued)
public class JDBCDemo
{
    final static String URL1 = "jdbc:derby:employee;create=true";
    final static String URL2 = "jdbc:sqlite:employee";

    public static void main(String[] args)
    {
        String url = null;
        if (args.length != 1)
        {
            System.err.println("usage 1: java JDBCDemo javadb");
            System.err.println("usage 2: java JDBCDemo sqlite");
            return;
        }
        if (args[0].equals("javadb"))
            url = URL1;
        else
            if (args[0].equals("sqlite"))
                url = URL2;
            else
            {
                System.err.println("invalid command-line argument");
                return;
            }
        Connection con = null;
        try
        {
            if (args[0].equals("sqlite"))
                Class.forName("org.sqlite.JDBC");
            con = DriverManager.getConnection(url);
            Statement stmt = null;
            try
            {
                stmt = con.createStatement();
                String sql = "CREATE TABLE EMPLOYEES(ID INTEGER, NAME VARCHAR(30))";
                stmt.executeUpdate(sql);
                PreparedStatement pstmt = null;
                try
                {
                    pstmt = con.prepareStatement("INSERT INTO EMPLOYEES VALUES(?, ?)");
                    String[] empNames = { "John Doe", "Sally Smith" };
                    for (int i = 0; i < empNames.length; i++)
                    {
                        pstmt.setInt(1, i + 1);
                        pstmt.setString(2, empNames[i]);
                        pstmt.executeUpdate();
                    }
                }
                ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEES");
                while (rs.next())
                    System.out.println(rs.getInt("ID") + " " + rs.getString("NAME"));
                stmt.executeUpdate("DROP TABLE EMPLOYEES");
            }
        }
    }
}
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
}
finally
{
    if (pstmt != null)
        try
            {
            pstmt.close();
            }
        catch (SQLException sqle)
            {
            sqle.printStackTrace();
            }
}
}
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
}
finally
{
    if (stmt != null)
        try
            {
            stmt.close();
            }
        catch (SQLException sqle)
            {
            sqle.printStackTrace();
            }
}
catch (ClassNotFoundException cnfe) {
    System.err.println("unable to load sqlite driver");
}
catch (SQLException sqlex) {
    while (sqlex != null) {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally {
    if (con != null) try {
        con.close();
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
}

Listing 14-3 creates a String object that specifies an SQL INSERT statement. Each “?” character serves as a placeholder for a value that’s specified before the statement is executed.

After the PreparedStatement implementation instance has been obtained, this interface’s void setInt(int parameterIndex, int x) and void setString(int parameterIndex, String x) methods are called on this instance to provide these values (the first argument passed to each method is a 1-based integer column index into the table associated with the statement—1 corresponds to the leftmost column), and then PreparedStatement’s int executeUpdate() method is called to execute this SQL statement. The end result is that a pair of rows containing John Doe, Sally Smith, and their respective identifiers is added to the EMPLOYEES table.

**CallableStatement**

CallableStatement is the most specialized of the statement interfaces; it extends PreparedStatement. You use this interface to execute SQL stored procedures in which a stored procedure is a list of SQL statements that perform a specific task (such as fire an employee). Java DB differs from other RDBMSs in that a stored procedure’s body is implemented as a public static Java method. Furthermore, the class in which this method is declared must be public.
CHAPTER 14: Accessing Databases

Note  SQLite doesn’t support stored procedures.

You create a stored procedure by executing an SQL statement that typically begins with CREATE PROCEDURE and then continues with RDBMS-specific syntax. For example, the Java DB syntax for creating a stored procedure, as specified on the web page at http://db.apache.org/derby/docs/10.8/ref/rrefcreateprocedurestatement.html, is as follows:

```
CREATE PROCEDURE procedure-name ([ procedure-parameter [, procedure-parameter ]]*)
 [ procedure-element ]*
```

`procedure-name` is expressed as

```
[ schemaName .] SQL92Identifier
```

`procedure-parameter` is expressed as

```
[ { IN | OUT | INOUT } [ parameter-name ] ] DataType
```

`procedure-element` is expressed as

```
{
 | [ DYNAMIC ] RESULT SETS INTEGER
 | LANGUAGE { JAVA }
 | DeterministicCharacteristic
 | EXTERNAL NAME string
 | PARAMETER STYLE JAVA
 | EXTERNAL SECURITY { DEFINER | INVOKER }
 | { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
}
```

Anything between [] is optional, the * to the right of [] indicates that anything between these metacharacters can appear zero or more times, the {} metacharacters surround a list of items, and | separates possible items—only one of these items can be specified.

For example, CREATE PROCEDURE FIRE(IN ID INTEGER) PARAMETER STYLE JAVA LANGUAGE JAVA DYNAMIC RESULT SETS 0 EXTERNAL NAME 'JDBCDemo.fire' creates a stored procedure named FIRE. This procedure specifies an input parameter named ID and is associated with a public static method named fire in a public class named JDBCDemo.

After creating the stored procedure, you need to obtain a CallableStatement implementation instance in order to call that procedure, and you do so by invoking one of Connection’s prepareCall() methods; for example, CallableStatement prepareCall(String sql).

The string passed to prepareCall() is an escape clause (RDBMS-independent syntax) consisting of an open {, followed by the word call, followed by a space, followed by the name of the stored procedure, followed by a parameter list with “?” placeholder characters for the arguments that will be passed, followed by a closing }.

www.it-ebooks.info
Note  Escape clauses are JDBC’s way of smoothing out some of the differences in how different RDBMS vendors implement SQL. When a JDBC driver detects escape syntax, it converts it into the code that the particular RDBMS understands. This makes escape syntax RDBMS independent.

Once you have a CallableStatement reference, you pass arguments to these parameters in the same way as with PreparedStatement. The following example demonstrates:

```java
CallableStatement cstmt = null;
try {
    cstmt = con.prepareCall("{ call FIRE(?)}")
    cstmt.setInt(1, 2);
    cstmt.execute();
} catch (SQLException sqle) {
    // handle the exception
}
finally {
    // close the callable statement
}
```

The `cstmt.setInt(1, 2)` method call assigns 2 to the leftmost stored procedure parameter—parameter index 1 corresponds to the leftmost parameter (or to a single parameter when there’s only one). The `cstmt.execute()` method call executes the stored procedure, which results in a callback to the application’s `public static void fire(int id)` method.

I’ve created another version of the JDBCDemo application that demonstrates this callable statement in a Java DB context only. Listing 14-4 presents its source code.

Listing 14-4.  Firing an Employee via a Stored Procedure

```java
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo {
    public static void main(String[] args) {
        String url = "jdbc:derby:employee;create=true";
        Connection con = null;
        try
```
```java
{ 
    con = DriverManager.getConnection(url);
    Statement stmt = null;
    try {
        stmt = con.createStatement();
        String sql = "CREATE PROCEDURE FIRE(IN ID INTEGER) " +
            "   PARAMETER STYLE JAVA" +
            "   LANGUAGE JAVA" +
            "   DYNAMIC RESULT SETS 0" +
            "   EXTERNAL NAME 'JDBCDemo.fire'";
        stmt.executeUpdate(sql);
        sql = "CREATE TABLE EMPLOYEES(ID INTEGER, NAME VARCHAR(30), " +
            "FIRED BOOLEAN)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO EMPLOYEES VALUES(1, 'John Doe', false)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO EMPLOYEES VALUES(2, 'Sally Smith', false)";
        stmt.executeUpdate(sql);
        dump(stmt.executeQuery("SELECT * FROM EMPLOYEES"));
        CallableStatement cstmt = null;
        try {
            cstmt = con.prepareCall("{ call FIRE(?)}");
            cstmt.setInt(1, 2);
            cstmt.execute();
            dump(stmt.executeQuery("SELECT * FROM EMPLOYEES"));
            stmt.executeUpdate("DROP TABLE EMPLOYEES");
            stmt.executeUpdate("DROP PROCEDURE FIRE");
        } catch (SQLException sqlex) {
            while (sqlex != null) {
                System.err.println("SQL error : " + sqlex.getMessage());
                System.err.println("SQL state : " + sqlex.getSQLState());
                System.err.println("Error code: " + sqlex.getErrorCode());
                System.err.println("Cause: " + sqlex.getCause());
                sqlex = sqlex.getNextException();
            }
        } finally {
            if (cstmt != null)
                try {
                    cstmt.close();
                }
        }
    }
} 
```
catch (SQLException sqle)
{
    sqle.printStackTrace();
}
}
} catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally
{
    if (stmt != null)
    try
    {
        stmt.close();
    } catch (SQLException sqle)
    {
        sqle.printStackTrace();
    }
}
} catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally
{
    if (con != null)
    try
    {
        con.close();
    }
catch (SQLException sqle)
{
    sqle.printStackTrace();
}

static void dump(ResultSet rs) throws SQLException
{
    StringBuilder sb = new StringBuilder();
    while (rs.next())
    {
        sb.append(rs.getInt("ID"));
        sb.append(' ');
        sb.append(rs.getString("NAME"));
        sb.append(' ');
        sb.append(rs.getBoolean("FIRED"));
        System.out.println(sb);
        sb.setLength(0);
    }
    System.out.println();
}

public static void fire(int id) throws SQLException
{
    Connection con = DriverManager.getConnection("jdbc:default:connection");
    String sql = "UPDATE EMPLOYEES SET FIRED=TRUE WHERE ID=\n" + id;
    Statement stmt = null;
    try
    {
        stmt = con.createStatement();
        stmt.executeUpdate(sql);
    }
    finally
    {
        if (stmt != null)
            try
            {
                stmt.close();
            }
            catch (SQLException sqle)
            {
                sqle.printStackTrace();
            }
    }
}

Much of 14-4 should be fairly understandable so I'll only discuss the fire() method. As previously stated, this method is invoked as a result of the callable statement invocation.
CHAPTER 14: Accessing Databases

fire() is called with the integer identifier of the employee to fire. It first accesses the current Connection object by invoking getConnection() with the jdbc.default:connection argument, which is supported by Oracle virtual machines through a special internal driver.

After creating an SQL UPDATE statement string to set the FIRED column to true in the EMPLOYEES table row where its ID field equals the value in id, fired() invokes executeUpdate() to update the table appropriately.

Compile Listing 14-4 (javac JDBCDemo.java) and run this application (java JDBCDemo). You should observe the following output:

1 John Doe false
2 Sally Smith false
1 John Doe false
2 Sally Smith true

Metadata

A data source is typically associated with metadata (data about data) that describes the data source. When the data source is an RDBMS, this data is typically stored in a collection of tables.

Metadata includes a list of catalogs (RDBMS databases whose tables describe RDBMS objects such as base tables [tables that physically exist], views [virtual tables], and indexes [files that improve the speed of data retrieval operations]), schemas (namespaces that partition database objects), and additional information (such as version numbers, identification strings, and limits).

To access a data source’s metadata, invoke Connection’s DatabaseMetaData getMetaData() method. This method returns an implementation instance of the java.sql.DatabaseMetaData interface.

I’ve created yet another JDBCDemo application that demonstrates getMetaData() and various DatabaseMetaData methods in the context of Java DB. Listing 14-5 presents MetaData’s source code.

Listing 14-5. Obtaining Metadata from an employee Data Source

```java
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo
{
    public static void main(String[] args)
    {
        String url = "jdbc:derby:employee;create=true";
        Connection con = null;
        try
        {
            con = DriverManager.getConnection(url);
            dump(con.getMetaData());
            dump(con.getMetaData());
        }
    }
```
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally
{
    if (con != null)
    try
    {
        con.close();
    } catch (SQLException sqle)
    {
        sqle.printStackTrace();
    }
}

static void dump(DatabaseMetaData dbmd) throws SQLException
{
    System.out.println("DB Major Version = " + dbmd.getDatabaseMajorVersion());
    System.out.println("DB Minor Version = " + dbmd.getDatabaseMinorVersion());
    System.out.println("DB Product = " + dbmd.getDatabaseProductName());
    System.out.println("Driver Name = " + dbmd.getDriverName());
    System.out.println("Numeric function names for escape clause = " +
                      dbmd.getNumericFunctions());
    System.out.println("String function names for escape clause = " +
                      dbmd.getStringFunctions());
    System.out.println("System function names for escape clause = " +
                      dbmd.getSystemFunctions());
    System.out.println("Time/date function names for escape clause = " +
                      dbmd.getTimeDateFunctions());
    System.out.println("Catalog term: " + dbmd.getCatalogTerm());
    System.out.println("Schema term: " + dbmd.getSchemaTerm());
    System.out.println();

    System.out.println("Catalogs");
    System.out.println("--------");
    ResultSet rsCat = dbmd.getCatalogs();
    while (rsCat.next())
    {
        System.out.println(rsCat.getString("TABLE_CAT"));
    }
    System.out.println();

    System.out.println("Schemas");
    System.out.println("-------");
    ResultSet rsSchem = dbmd.getSchemas();
    System.out.println();
    System.out.println();
while (rsSchem.next())
    System.out.println(rsSchem.getString("TABLE_SCHEM"));
System.out.println();
System.out.println("Schema/Table");
System.out.println("-----------");
rsSchem = dbmd.getSchemas();
while (rsSchem.next())
{
    String schem = rsSchem.getString("TABLE_SCHEM");
    ResultSet rsTab = dbmd.getTables(null, schem, "%", null);
    while (rsTab.next())
        System.out.println(schem + " " + rsTab.getString("TABLE_NAME"));
}

Listing 14-5’s dump() method invokes various methods on its dbmd argument to output assorted metadata.

The int getDatabaseMajorVersion() and int getDatabaseMinorVersion() methods return the major (such as 10) and minor (such as 8) parts of Java DB’s version number. Similarly, String getDatabaseProductName() returns the name of this product (such as Apache Derby), and String getDriverName() returns the name of the driver (such as Apache Derby Embedded JDBC Driver).

SQL defines various functions that can be invoked as part of SELECT and other statements. For example, you can specify SELECT COUNT(*) AS TOTAL FROM EMPLOYEES to return a one-row-by-one-column result set with the column named TOTAL and the row value containing the number of rows in the EMPLOYEES table.

Because not all RDMSes adopt the same syntax for specifying function calls, JDBC uses a function escape clause, consisting of \{ fn functionname(arguments) \}, to abstract over differences. For example, SELECT \{ fn UCASE(NAME) \} FROM EMPLOYEES selects all NAME column values from EMPLOYEES and uppercases their values in the result set.

The String getNumericFunctions(), String getStringFunctions(), String getSystemFunctions(), and String getTimeDateFunctions() methods return lists of function names that can appear in function escape clauses. For example, getNumericFunctions() returns ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, MOD, PI, RADIANS, RAND, SIGN, SIN, SQRT, TAN for Java DB 10.8.

Not all vendors use the same terminology for catalog and schema. For this reason, the String getCatalogTerm() and String getSchemaTerm() methods are present to return the vendor-specific terms, which happen to be CATALOG and SCHEMA for Java DB 10.8.

The ResultSet getCatalogs() method returns a result set of catalog names, which are accessible via the result set’s TABLE_CAT column. This result set is empty for Java DB 10.8, which divides a single default catalog into various schemas.

The ResultSet getSchemas() method returns a result set of schema names, which are accessible via the result set’s TABLE_SCHEM column. This column contains APP, NULLID, SQLJ, SYS, SYSCAT, SYSCS_DIAG, SYSCS_UTIL, SYSFUN, SYSIBM, SYSPROC, and SYSSTAT values for Java DB 10.8. APP is the default schema in which a user’s database objects are stored.
The ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) method returns a result set containing table names (in the TABLE_NAME column) and other table-oriented metadata that match the specified catalog, schemaPattern, tableNamePattern, and types. To obtain a result set of all tables for a specific schema, pass null to catalog and types, the schema name to schemaPattern, and the % wildcard to tableNamePattern.

For example, the SYS schema stores SYSALIASES, SYSCHECKS, SYSCOLPERMS, SYSCOLUMNS, SYSCONGLOMERATES, SYSCONSTRAINTS, SYSDEPENDS, SYSFILES, SYSFOREIGNKEYS, SYSKEYS, SYSPERMS, SYSSCHEMA, SYSROUTEINPERMS, SYSSEQUENCES, SYSSTATEMENTS, SYSSTATISTICS, SYSTABLEPERMS, SYSTABLES, SYSTRIGGERS, and SYSVIEWS tables.

Listings 14-2 through 14-4 suffer from an architectural problem. After creating the EMPLOYEES table, suppose that SQLException is thrown before the table is dropped. The next time the JDBCDemo application runs (under Java DB), SQLException is thrown when the application attempts to recreate EMPLOYEES because this table already exists. You have to manually delete the employee directory before you can rerun JDBCDemo.

It would be nice to call an isExist() function before creating EMPLOYEES, but that function doesn’t exist. However, you can create a same-named method with help from getTables(), and Listing 14-6 shows you how to accomplish this task.

Listing 14-6. Determining the Existence of EMPLOYEES Before Creating This Table

```java
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo
{
    public static void main(String[] args)
    {
        String url = "jdbc:derby:employee;create=true";
        Connection con = null;
        try
        {
            con = DriverManager.getConnection(url);
            Statement stmt = null;
            try
            {
                stmt = con.createStatement();
                String sql;
                if (!isExist(con, "EMPLOYEES"))
                {
                    System.out.println("EMPLOYEES doesn't exist");
                    sql = "CREATE TABLE EMPLOYEES(ID INTEGER, NAME VARCHAR(30))";
                    stmt.executeUpdate(sql);
                }
            }
        }
    }
}
```
else
    System.out.println("EMPLOYEES already exists");
sql = "INSERT INTO EMPLOYEES VALUES(1, 'John Doe')";
stmt.executeUpdate(sql);
sql = "INSERT INTO EMPLOYEES VALUES(2, 'Sally Smith')";
stmt.executeUpdate(sql);
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEES");
while (rs.next())
    System.out.println(rs.getInt("ID") + " " + rs.getString("NAME");
stmt.executeUpdate("DROP TABLE EMPLOYEES");
}
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
}
finally
{
    if (stmt != null)
        try
        {
            stmt.close();
        }
        catch (SQLException sqle)
        {
            sqle.printStackTrace();
        }
}
catch (SQLException sqlex)
{
    while (sqlex != null)
    {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
}
finally
{
    if (con != null)
        try

Listing 14-6 refactors Listing 14-2 (from a Java DB perspective only) by introducing a boolean isExist(Connection con, String tableName) class method, which returns true when tableName exists, and using this method to determine the existence of EMPLOYEES before creating this table. When the specified table exists, a ResultSet object containing one row is returned, and ResultSet’s next() method returns true. Otherwise, the result set contains no rows and next() returns false.

Caution  isExist() assumes the default APP schema, which might not be the case when usernames are involved (each user’s database objects are stored in a schema corresponding to the user’s name).

EXERCISES

The following exercises are designed to test your understanding of Chapter 14’s content.

1. Define database.
2. What is a relational database?
3. Identify two other database categories.
4. Define database management system.
5. What is Java DB?
6. True or false: Java DB’s client driver causes the database engine to run in the same virtual machine as the application.
7. What does setEmbeddedCP accomplish?
8. True or false: You run Java DB’s dblook command-line tool to view the Java environment/Java DB configuration.
9. What is SQLite?
10. Define manifest typing.
11. What tool does SQLite provide for accessing and modifying SQLite databases?
12. What is JDBC?
14. A JDBC driver implements what interface?
15. True or false: There are three kinds of JDBC drivers.
16. Describe a Type 3 JDBC driver.
17. What types does JDBC provide for communicating with a data source?
18. How do you obtain a connection to a Java DB data source via the embedded driver?
19. True or false: String getSQLState() returns a vendor-specific error code.
20. What is a SQL state error code?
21. What is the difference between SQLNonTransientException and SQLTransientException?
22. Identify JDBC’s three statement types.
23. Which Statement method do you call to execute an SQL SELECT statement?
24. What does a result set’s cursor accomplish?
25. To which Java type does the SQL FLOAT type map?
26. What does a prepared statement represent?
27. True or false: CallableStatement extends PreparedStatement.
29. How do you call a stored procedure?
30. What is an escape clause?
31. Define metadata.
32. What does metadata include?
33. Refactor Listing 14-5 to output metadata for the SQLite driver as well as for the Java DB embedded driver.
Summary

A database is an organized collection of data. Although there are many kinds of databases (such as hierarchical, object-oriented, and relational), relational databases, which organize data into tables that can be related to each other, are common.

Except for the most trivial of databases (such as Chapter 11’s flat file database based on a single data file), databases are created and managed through a database management system. Relational DBMSs support SQL for working with tables and more.

First introduced by Sun Microsystems as part of JDK 6 (and not included in the JRE) to give developers an RDBMS to test their JDBC code, Java DB is a distribution of Apache’s open-source Derby product, which is based on IBM’s Cloudscape RDBMS code base.

Java DB is capable of running in an embedded environment or in a client/server environment. In an embedded environment, where an application accesses the database engine via Java DB’s embedded driver, the database engine runs in the same virtual machine as the application.

In a client/server environment, client applications and the database engine run in separate virtual machines. A client application accesses the network server through Java DB’s client driver. The network server, which runs in the same virtual machine as the database engine, accesses the database engine through the embedded driver.

SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine; it is considered to be the most widely deployed database engine in the world. For example, SQLite is found in Mozilla Firefox, Google Chrome, and other web browsers. It’s also found in Google Android, Apple iOS, and other mobile operating systems.

The sqlite3 executable offers a command-line shell for accessing and modifying SQLite databases. You can specify sqlite3 with a database filename argument to create the database file when it doesn’t exist (you must create a table at least) or open the existing file, and enter this tool’s shell from where you can execute sqlite3-specific, dot-prefixed commands and SQL statements.

JDBC is an API for performing various database operations, such as submitting SQL statements that tell the RDBMS to create a table and to update or query tabular data. Although JDBC is typically used to communicate with RDBMSs, it also can be used to communicate with a flat file database. For this reason, JDBC uses the term data source to abstract the source of data.

Because data sources are accessed in different ways, JDBC uses drivers to abstract over their implementations. This abstraction lets you write an application that can be adapted to an arbitrary data source without having to change a single line of code (in most cases). Drivers are implementations of the java.sql.Driver interface. JDBC recognizes four types of drivers.

To connect to a data source and obtain a Connection instance, call one of DriverManager’s getConnection() methods. With either method, the url argument specifies a string-based URL that starts with the jdbc: prefix and continues with data source-specific syntax.

DriverManager’s getConnection() methods (and other JDBC methods in the various JDBC interfaces) throw SQLException or one of its subclasses when something goes wrong. Instances of this class provide vendor codes, SQL state strings, and other kinds of information.
After obtaining a connection to a data source, an application interacts with the data source by issuing SQL statements. JDBC supports SQL statements via the Statement, PreparedStatement, and CallableStatement interfaces.

The `executeQuery()` methods return a ResultSet object that provides access to a query’s tabular results. Each result set is associated with a cursor that provides access to a specific row of data. The cursor initially points before the first row.

ResultSet also declares various methods for returning the current row’s column values based on their types. For example, `int getInt(String columnLabel)` returns the integer value corresponding to the INTEGER-based column identified by `columnLabel`.

A prepared statement represents a precompiled SQL statement. The SQL statement is compiled to improve performance and prevent SQL injection, and the compiled result is stored in a PreparedStatement implementation instance.

A callable statement is a special kind of prepared statement for executing SQL stored procedures in which a stored procedure is a list of SQL statements that perform a specific task. The argument passed to a callable statement’s `prepareCall()` method is specified using escape syntax.

A data source is typically associated with metadata that describes the data source. When the data source is an RDBMS, this data is typically stored in a collection of tables. Metadata includes a list of catalogs, base tables, views, indexes, schemas, and additional information.

Databases can store XML documents, which are a convenient way to exchange data. Chapter 15 introduces you to XML and shows you how to parse, create, and transform XML documents.
Chapter 15

Parsing, Creating, and Transforming XML Documents

Applications commonly use XML documents to store and exchange data. In Chapter 15, I introduce XML for the benefit of those who are unfamiliar with this technology.

Java supports XML via the SAX, DOM, StAX, XPath, and XSLT APIs. After introducing XML, I also introduce these APIs, except for StAX, which Android doesn’t support. Instead of StAX, I introduce Android’s XMLPULL V1 API, which is somewhat equivalent to StAX.

What Is XML?

XML (eXtensible Markup Language) is a metalanguage (a language used to describe other languages) for defining vocabularies (custom markup languages), which is key to XML’s importance and popularity. XML-based vocabularies (such as XHTML) let you describe documents in a meaningful way.

XML vocabulary documents are like HTML (see http://en.wikipedia.org/wiki/HTML) documents in that they are text-based and consist of markup (encoded descriptions of a document’s logical structure) and content (document text not interpreted as markup). Markup is evidenced via tags (angle bracket-delimited syntactic constructs) and each tag has a name. Furthermore, some tags have attributes (name-value pairs).
Note XML and HTML are descendants of Standard Generalized Markup Language (SGML), which is the original metalanguage for creating vocabularies. XML is essentially a restricted form of SGML, while HTML is an application of SGML. The key difference between XML and HTML is that XML invites you to create your own vocabularies with their own tags and rules, whereas HTML gives you a single precreated vocabulary with its own fixed set of tags and rules. XHTML and other XML-based vocabularies are XML applications. XHTML was created to be a cleaner implementation of HTML.

If you haven’t previously encountered XML, you might be surprised by its simplicity and how closely its vocabularies resemble HTML. You don’t need to be a rocket scientist to learn how to create an XML document. To prove this to yourself, check out Listing 15-1.

Listing 15-1. XML-Based Recipe for a Grilled Cheese Sandwich

```xml
<recipe>
  <title>
    Grilled Cheese Sandwich
  </title>
  <ingredients>
    <ingredient qty="2">
      bread slice
    </ingredient>
    <ingredient>
      cheese slice
    </ingredient>
    <ingredient qty="2">
      margarine pat
    </ingredient>
  </ingredients>
  <instructions>
    Place frying pan on element and select medium heat. For each bread slice, smear one pat of margarine on one side of bread slice. Place cheese slice between bread slices with margarine-smeared sides away from the cheese. Place sandwich in frying pan with one margarine-smeared side in contact with pan. Fry for a couple of minutes and flip. Fry other side for a minute and serve.
  </instructions>
</recipe>
```

Listing 15-1 presents an XML document that describes a recipe for making a grilled cheese sandwich. This document is reminiscent of an HTML document in that it consists of tags, attributes, and content. However, that’s where the similarity ends. Instead of presenting HTML tags such as `<html>`, `<head>`, `<img>`, and `<p>`, this informal recipe language presents its own `<recipe>`, `<ingredients>`, and other tags.
Note Although Listing 15-1’s <title> and </title> tags are also found in HTML, they differ from their HTML counterparts. Web browsers typically display the content between these tags in their title bars. In contrast, the content between Listing 15-1’s <title> and </title> tags might be displayed as a header, spoken aloud, or presented in some other way, depending on the application that parses this document.

XML documents are based on the XML declaration, elements and attributes, character references and CDATA sections, namespaces, and comments and processing instructions. After learning about these fundamentals, you’ll learn what it means for an XML document to be well formed. You will also learn what it means for an XML document to be valid.

XML Declaration

An XML document will typically begin with the XML declaration, special markup that informs an XML parser that the document is XML. The absence of the XML declaration in Listing 15-1 reveals that this special markup isn’t mandatory. When the XML declaration is present, nothing can appear before it.

The XML declaration minimally looks like <?xml version="1.0"?> in which the nonoptional version attribute identifies the version of the XML specification to which the document conforms. The initial version of this specification (1.0) was introduced in 1998 and is widely implemented.

Note The World Wide Web Consortium (W3C), which maintains XML, released version 1.1 in 2004. This version mainly supports the use of line-ending characters used on EBCDIC platforms (see http://en.wikipedia.org/wiki/EBCDIC) and the use of scripts and characters that are absent from Unicode 3.2 (see http://en.wikipedia.org/wiki/Unicode). Unlike XML 1.0, XML 1.1 isn’t widely implemented and should be used only by those needing its unique features.

XML supports Unicode, which means that XML documents consist entirely of characters taken from the Unicode character set. The document’s characters are encoded into bytes for storage or transmission, and the encoding is specified via the XML declaration’s optional encoding attribute. One common encoding is UTF-8 (see http://en.wikipedia.org/wiki/UTF-8), which is a variable-length encoding of the Unicode character set. UTF-8 is a strict superset of ASCII (see http://en.wikipedia.org/wiki/Ascii), which means that pure ASCII text files are also UTF-8 documents.
If you’ll never use characters apart from the ASCII character set, you can probably forget about the encoding attribute. However, when your native language isn’t English or when you’re called to create XML documents that include non-ASCII characters, you need to specify encoding properly. For example, when your document contains ASCII plus characters from a non-English Western European Language (such as ç, the cedilla used in French, Portuguese, and other languages), you might want to choose ISO-8859-1 as the encoding attribute’s value; the document will probably have a smaller size when encoded in this manner than when encoded with UTF-8. Listing 15-2 shows you the resulting XML declaration.

Listing 15-2. An Encoded Document Containing Non-ASCII Characters

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<movie>
  <name>Le Fabuleux Destin d'Amélie Poulain</name>
  <language>français</language>
</movie>
```

The final attribute that can appear in the XML declaration is standalone. This optional attribute determines whether the XML document relies on an external DTD (discussed later in this chapter) or not: its value is no when relying on an external DTD, or yes when not relying on an external DTD. The value defaults to no, implying that there is an external DTD. However, because there’s no guarantee of a DTD, standalone is rarely used and won’t be discussed further.

**Elements and Attributes**

Following the XML declaration is a hierarchical (tree) structure of elements, where an element is a portion of the document delimited by a start tag (such as `<name>`) and an end tag (such as `</name>`), or is an empty-element tag (a standalone tag whose name ends with a forward slash `/`), such as `<break/>`. Start tags and end tags surround content and possibly other markup whereas empty-element tags don’t surround anything. Figure 15-1 reveals Listing 15-1’s XML document tree structure.
As with HTML document structure, the structure of an XML document is anchored in a root element (the topmost element). In HTML, the root element is `html` (the `<html>` and `</html>` tag pair). Unlike in HTML, you can choose the root element for your XML documents. Figure 15-1 shows the root element to be `recipe`.

Unlike the other elements that have parent elements, `recipe` has no parent. Also, `recipe` and `ingredients` have child elements: `recipe`'s children are `title`, `ingredients`, and `instructions`; and `ingredients`' children are three instances of `ingredient`. The `title`, `instructions`, and `ingredient` elements don't have child elements.

Elements can contain child elements, content, or mixed content (a combination of child elements and content). Listing 15-2 reveals that the `movie` element contains `name` and `language` child elements, and it also reveals that each of these child elements contains content (`language` contains `français`, for example). Listing 15-3 presents another example that demonstrates mixed content along with child elements and content.

Listing 15-3. An abstract Element Containing Mixed Content

```xml
<?xml version="1.0"?>
<article title="The Rebirth of JavaFX" lang="en">
  <abstract>
    JavaFX 2.0 marks a significant milestone in the history of JavaFX. Now that Sun Microsystems has passed the torch to Oracle, we have seen the demise of JavaFX Script and the emerge of Java APIs (such as `javafx.application.Application`) for interacting with this technology. This article introduces you to this new flavor of JavaFX, where you learn about JavaFX 2.0 architecture and key APIs.
  </abstract>
  <body>
    </body>
</article>
```

This document's root element is `article`, which contains `abstract` and `body` child elements. The `abstract` element mixes content with a `code-inline` element, which contains content. In contrast, the `body` element is empty.
Note As with Listings 15-1 and 15-2, Listing 15-3 also contains whitespace (invisible characters such as spaces, tabs, carriage returns, and line feeds). The XML specification permits whitespace to be added to a document. Whitespace appearing within content (such as spaces between words) is considered part of the content. In contrast, the parser typically ignores whitespace appearing between an end tag and the next start tag. Such whitespace isn’t considered part of the content.

An XML element’s start tag can contain one or more attributes. For example, Listing 15-1’s `<ingredient>` tag has a `qty` (quantity) attribute and Listing 15-3’s `<article>` tag has `title` and `lang` attributes. Attributes provide additional information about elements. For example, `qty` identifies the amount of an ingredient that can be added, `title` identifies an article’s title, and `lang` identifies the language in which the article is written (en for English). Attributes can be optional. For example, when `qty` isn’t specified, a default value of 1 is assumed.

Note Element and attribute names may contain any alphanumeric character from English or another language, and they may also include the underscore (_), hyphen (-), period (.), and colon (:)) punctuation characters. The colon should only be used with namespaces (discussed later in this chapter), and names cannot contain whitespace.

Character References and CDATA Sections

Certain characters cannot appear literally in the content that appears between a start tag and an end tag, or within an attribute value. For example, you cannot place a literal `<` character between a start tag and an end tag because doing so would confuse an XML parser into thinking that it had encountered another tag.

One solution to this problem is to replace the literal character with a character reference, which is a code that represents the character. Character references are classified as numeric character references or character entity references.

- A numeric character reference refers to a character via its Unicode code point and adheres to the format `&#nnnn;` (not restricted to four positions) or `&#xhhhh;` (not restricted to four positions), where `nnnn` provides a decimal representation of the code point and `hhhh` provides a hexadecimal representation. For example, `&#0931;` and `&#x03A3;` represent the Greek capital letter sigma. Although XML mandates that the `x` in `&#xhhhh;` be lowercase, it's flexible in that the leading zero is optional in either format and in allowing you to specify an uppercase or lowercase letter for each `h`. As a result, `&#931;`, `&#x3A3;`, and `&#x03a3;` are also valid representations of the Greek capital letter sigma.
A character entity reference refers to a character via the name of an entity (aliased data) that specifies the desired character as its replacement text. Character entity references are predefined by XML and have the format &name;, in which name is the entity’s name. XML predefines five character entity references: &lt (,), &gt (>), &amp (&), &apos ('), and &quot; (").

Consider <expression>6 < 4</expression>. You could replace the < with numeric reference &#60;, yielding <expression>6 &lt; 4</expression>, or better yet with &lt; yielding <expression>6 &lt; 4</expression>. The second choice is clearer and easier to remember.

Suppose you want to embed an HTML or XML document within an element. To make the embedded document acceptable to an XML parser, you would need to replace each literal < (start of tag) and & (start of entity) character with its < and & predefined character entity reference, a tedious and possibly error prone undertaking since you might forget to replace one of these characters. To save you from tedium and potential errors, XML provides an alternative in the form of a CDATA (character data) section.

A CDATA section is a section of literal HTML or XML markup and content surrounded by the <![CDATA[ prefix and the ]]> suffix. You don’t need to specify predefined character entity references within a CDATA section, as demonstrated in Listing 15-4.

Listing 15-4. Embedding an XML Document in Another Document’s CDATA Section

```xml
<?xml version="1.0"?>
<svg-examples>
  <example>
    The following Scalable Vector Graphics document describes a blue-filled and black-stroked rectangle. <![CDATA[<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
  <rect width="300" height="100"
    style="fill:rgb(0,0,255);stroke-width:1; stroke:rgb(0,0,0")/>
  </svg>]]>
  </example>
</svg-examples>
```

Listing 15-4 embeds a Scalable Vector Graphics (SVG) [see http://en.wikipedia.org/wiki/Svg] XML document within the example element of an SVG examples document. The SVG document is placed in a CDATA section, obviating the need to replace all < characters with &lt; predefined character entity references.

Namespaces

It’s common to create XML documents that combine features from different XML languages. Namespaces are used to prevent name conflicts when elements and other XML language features appear. Without namespaces, an XML parser couldn’t distinguish between same-named elements or other language features that mean different things, such as two same-named title elements from two different languages.
Note Namespaces aren’t part of XML 1.0. They arrived about a year after this specification was released. To ensure backward compatibility with XML 1.0, namespaces take advantage of colon characters, which are legal characters in XML names. Parsers that don’t recognize namespaces return names that include colons.

A namespace is a Uniform Resource Identifier (URI)-based container that helps differentiate XML vocabularies by providing a unique context for its contained identifiers. The namespace URI is associated with a namespace prefix (an alias for the URI) by specifying, typically on an XML document’s root element, either the xmlns attribute by itself (which signifies the default namespace) or the xmlns:prefix attribute (which signifies the namespace identified as prefix), and assigning the URI to this attribute.

Note A namespace’s scope starts at the element where it’s declared and is applied to all of the element’s content unless overridden by another namespace declaration with the same prefix name.

When prefix is specified, it and a colon character are prepended to the name of each element tag that belongs to that namespace (see Listing 15-5).

Listing 15-5. Introducing a Pair of Namespaces

```xml
<?xml version="1.0"?>
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
       xmlns:r="http://www.tutortutor.ca/"
       xmlns:xmlns=
<h:head>
  <h:title>
    Recipe
  </h:title>
</h:head>
<h:body>
<r:recipe>
  <r:title>
    Grilled Cheese Sandwich
  </r:title>
  <r:ingredients>
    <h:ul>
      <h:li>
        <r:ingredient qty="2">
          bread slice
        </r:ingredient>
      </h:li>
      <h:li>
        <r:ingredient>
          cheese slice
        </r:ingredient>
      </h:li>
    </h:ul>
  </r:ingredients>
</r:recipe>
</h:body>
```

www.it-ebooks.info
Listing 15-5 describes a document that combines elements from the XHTML language (see http://en.wikipedia.org/wiki/XHTML) with elements from the recipe language. All element tags that associate with XHTML are prefixed with h:, and all element tags that associate with the recipe language are prefixed with r:.

The h: prefix associates with the www.w3.org/1999/xhtml URI, and the r: prefix associates with the www.tutortutor.ca URI. XML doesn’t mandate that URIs point to document files. It only requires that they be unique to guarantee unique namespaces.

This document’s separation of the recipe data from the XHTML elements makes it possible to preserve this data’s structure while also allowing an XHTML-compliant web browser (such as Google Chrome) to present the recipe via a web page (see Figure 15-2).

Figure 15-2. Google Chrome presents the recipe data via XHTML tags
A tag's attributes don't need to be prefixed when those attributes belong to the element. For example, qty isn't prefixed in `<r:ingredient qty="2">`. However, a prefix is required for attributes belonging to other namespaces. For example, suppose you want to add an XHTML style attribute to the document's `<r:title>` tag to provide styling for the recipe title when displayed via an application. You can accomplish this task by inserting an XHTML attribute into the title tag, as follows:

```xml
<r:title h:style="font-family: sans-serif;">
```

The XHTML style attribute has been prefixed with h: because this attribute belongs to the XHTML language namespace and not to the recipe language namespace.

When multiple namespaces are involved, it can be convenient to specify one of these namespaces as the default namespace to reduce the tedium in entering namespace prefixes. Consider Listing 15-6.

**Listing 15-6. Specifying a Default Namespace**

```xml
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:r="http://www.tutortutor.ca/">
  <head>
    <title>Recipe</title>
  </head>
  <body>
    <r:recipe>
      <r:title>Grilled Cheese Sandwich</r:title>
      <r:ingredients>
        <ul>
          <li>:ingredient qty="2">
            bread slice
          </li>
          <li>:ingredient>
            cheese slice
          </li>
          <li>:ingredient qty="2">
            margarine pat
          </li>
        </ul>
      </r:ingredients>
      <p>
      <r:instructions>
```
Place frying pan on element and select medium heat. For each bread slice, smear one pat of margarine on one side of bread slice. Place cheese slice between bread slices with margarine-smeared sides away from the cheese. Place sandwich in frying pan with one margarine-smeared side in contact with pan. Fry for a couple of minutes and flip. Fry other side for a minute and serve.

Listing 15-6 specifies a default namespace for the XHTML language. No XHTML element tag needs to be prefixed with `h:`. However, recipe language element tags must still be prefixed with the `r:` prefix.

**Comment and Processing Instructions**

XML documents can contain *comments*, which are character sequences beginning with `<!--` and ending with `-->`. For example, you might place `<!-- Todo -->` in Listing 15-3’s body element to remind yourself that you need to finish coding this element.

Comments are used to clarify portions of a document. They can appear anywhere after the XML declaration except within tags; they cannot be nested, cannot contain a double hyphen (`--`) because doing so might confuse an XML parser that the comment has been closed, shouldn’t contain a hyphen (`-`) for the same reason, and they are typically ignored during processing. Comments are not content.

XML also permits processing instructions to be present. A *processing instruction* is an instruction that’s made available to the application parsing the document. The instruction begins with `<?` and ends with `?>`. The `<?` prefix is followed by a name known as the *target*. This name typically identifies the application to which the processing instruction is intended. The rest of the processing instruction contains text in a format appropriate to the application. Two examples of processing instructions are `<?xml-stylesheet href="modern.xsl" type="text/xml"?>` (associate an eXtensible Stylesheet Language [XSL] style sheet [see http://en.wikipedia.org/wiki/XSL] with an XML document) and `<?php /* PHP code */ ?>` (pass a PHP [see http://en.wikipedia.org/wiki/Php] code fragment to the application). Although the XML declaration looks like a processing instruction, this isn’t the case.

**Note**  The XML declaration isn’t a processing instruction.
Well-Formed Documents

HTML is a sloppy language in which elements can be specified out of order, end tags can be omitted, and so on. The complexity of a web browser’s page layout code is partly due to the need to handle these special cases. In contrast, XML is a much stricter language. To make XML documents easier to parse, XML mandates that XML documents follow certain rules:

- **All elements must either have start and end tags or consist of empty-element tags.** For example, unlike the HTML `<p>` tag that’s often specified without a `</p>` counterpart, `</p>` must also be present from an XML document perspective.

- **Tags must be nested correctly.** For example, while you’ll probably get away with specifying `<b><i>Android</i></b>` in HTML, an XML parser would report an error. In contrast, `<b><i>Android</i></b>` doesn’t result in an error.

- **All attribute values must be quoted.** Either single quotes (`'`) or double quotes (`"`) are permissible (although double quotes are the more commonly specified quotes). It’s an error to omit these quotes.

- **Empty elements must be properly formatted.** For example, HTML’s `<br>` tag would have to be specified as `<br/>` in XML. You can specify a space between the tag’s name and the `/` character, although the space is optional.

- **Be careful with case.** XML is a case-sensitive language in which tags differing in case (such as `<author>` and `<Author>`) are considered different. It’s an error to mix start and end tags of different cases, for example, `<author>` with `</Author>`.

XML parsers that are aware of namespaces enforce two additional rules:

- All element and attribute names must not include more than one colon character.

- No entity names, processing instruction targets, or notation names (discussed later) can contain colons.

An XML document that conforms to these rules is well formed. The document has a logical and clean appearance and is much easier to process. XML parsers will only parse well-formed XML documents.

Valid Documents

It’s not always enough for an XML document to be well formed; in many cases, the document must also be valid. A valid document adheres to constraints. For example, a constraint could be placed upon Listing 15-1’s recipe document to ensure that the `ingredients` element always precedes the `instructions` element; perhaps an application must first process `ingredients`.

**Note** XML document validation is similar to a compiler analyzing source code to make sure that the code makes sense in a machine context. For example, each of `int, count, =, 1, and ;` are valid Java character sequences but `1 count; int = isn’t a valid Java construct (whereas int count = 1; is a valid Java construct).
Some XML parsers perform validation, whereas other parsers don’t because validating parsers are harder to write. A parser that performs validation compares an XML document to a grammar document. Any deviation from this document is reported as an error to the application; the document isn’t valid. The application may choose to fix the error or reject the document. Unlike well-formedness errors, validity errors aren’t necessarily fatal and the parser can continue to parse the document.

Note Validating XML parsers often don’t validate by default because validation can be time-consuming. They must be instructed to perform validation.

Grammar documents are written in a special language. Two commonly-used grammar languages are Document Type Definition and XML Schema.

**Document Type Definition**

*Document Type Definition (DTD)* is the oldest grammar language for specifying an XML document’s grammar. DTD grammar documents (known as DTDs) are written in accordance with a strict syntax that states what elements may be present and in what parts of a document. It also states what is contained within elements (child elements, content, or mixed content) and what attributes may be specified. For example, a DTD may specify that a recipe element must have an ingredients element followed by an instructions element.

Listing 15-7 presents a DTD for the recipe language that was used to construct Listing 15-1’s document.

### Listing 15-7. The Recipe Language’s DTD

```xml
<!ELEMENT recipe (title, ingredients, instructions)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT ingredients (ingredient+)>
<!ELEMENT ingredient (#PCDATA)>
<!ELEMENT instructions (#PCDATA)>
<!ATTLIST ingredient qty CDATA "1">
```

This DTD first declares the recipe language’s elements. Element declarations take the form `<!ELEMENT name content-specifier>`, where `name` is any legal XML name (it cannot contain whitespace, for example), and `content-specifier` identifies what can appear within the element.

The first element declaration states that exactly one recipe element can appear in the XML document; this declaration doesn’t imply that recipe is the root element. Furthermore, this element must include exactly one each of the title, ingredients, and instructions child elements, and in that order. Child elements must be specified as a comma-separated list. Furthermore, a list is always surrounded by parentheses.

The second element declaration states that the title element contains *parsed character data* (nonmarkup text). The third element declaration states that at least one ingredient element must appear in ingredients. The `+` character is an example of a regular expression that means one or more. Other expressions that may be used are `*` (zero or more) and `?` (once or not at all). The fourth and fifth element declarations are similar to the second by stating that ingredient and instructions elements contain parsed character data.
**Note** Element declarations support three other content specifiers. You can specify `<!ELEMENT name ANY>` to allow any type of element content or `<!ELEMENT name EMPTY>` to disallow any element content. To state that an element contains mixed content, you would specify #PCDATA and a list of element names, separated by vertical bars (`|`). For example, `<!ELEMENT ingredient (#PCDATA | measure | note)*>` states that the `ingredient` element can contain a mix of parsed character data, zero or more `measure` elements, and zero or more `note` elements. It doesn’t specify the order in which the parsed character data and these elements occur. However, #PCDATA must be the first item specified in the list. When a regular expression is used in this context, it must appear to the right of the closing parenthesis.

Listing 15-7’s DTD lastly declares the recipe language’s attributes, of which there is only one: `qty`. Attribute declarations take the form `<!ATTLIST ename aname type default-value>`, where `ename` is the name of the element to which the attribute belongs, `aname` is the name of the attribute, `type` is the attribute’s type, and `default-value` is the attribute’s default value.

The attribute declaration identifies `qty` as an attribute of `ingredient`. It also states that `qty`’s type is CDATA (any string of characters not including the ampersand, less than or greater than signs, or double quotes may appear; these characters may be represented via `&`, `<`, `>`, or `"`, respectively), and that `qty` is optional, assuming default value 1 when not present.

### MORE ABOUT ATTRIBUTES

DTD lets you specify additional attribute types: ID (create a unique identifier for an attribute that identifies an element), IDREF (an attribute’s value is an element located elsewhere in the document), IDREFS (the value consists of multiple IDREFs), ENTITY (you can use external binary data or unparsed entities), ENTITIES (the value consists of multiple entities), NMTOKEN (the value is restricted to any valid XML name), NMTOKENS (the value is composed of multiple XML names), NOTATION (the value is already specified via a DTD notation declaration), and enumerated (a list of possible values from which to choose; values are separated with vertical bars).

Instead of specifying a default value verbatim, you can specify `#REQUIRED` to mean that the attribute must always be present with some value (`<!ATTLIST ename aname type #REQUIRED>`), `#IMPLIED` to mean that the attribute is optional and no default value is provided (`<!ATTLIST ename aname type #IMPLIED>`), or `#FIXED` to mean that the attribute is optional and must always take on the DTD-assigned default value when used (`<!ATTLIST ename aname type #FIXED "value">`).

You can specify a list of attributes in one ATTLIST declaration. For example, `<!ATTLIST ename aname1 type1 default-value1 aname2 type2 default-value2>` declares two attributes identified as `aname1` and `aname2`.

A DTD-based validating XML parser requires that a document include a `document type declaration` identifying the DTD that specifies the document’s grammar before it will validate the document.
A document type declaration appears immediately after the XML declaration, and it is specified in one of the following ways:

- `<!DOCTYPE root-element-name SYSTEM uri>` references an external but private DTD via `uri`. The referenced DTD isn’t available for public scrutiny. For example, I might store my recipe language’s DTD file (`recipe.dtd`) in a private `dtds` directory on my `www.tutortutor.ca` website and use `<!DOCTYPE recipe SYSTEM "http://www.tutortutor.ca/dtds/recipe.dtd">` to identify this DTD’s location via `system identifier` `http://www.tutortutor.ca/dtds/recipe.dtd`.

- `<!DOCTYPE root-element-name PUBLIC fpi uri>` references an external but public DTD via `fpi`, a formal public identifier (see [http://en.wikipedia.org/wiki/Formal_Public_Identifier](http://en.wikipedia.org/wiki/Formal_Public_Identifier)), and `uri`. If a validating XML parser cannot locate the DTD via public identifier `fpi`, it can use system identifier `uri` to locate the DTD. For example, `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">` references the XHTML 1.0 DTD first via public identifier `-//W3C//DTD XHTML 1.0 Transitional//EN` and second via system identifier `http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd`.

- `<!DOCTYPE root-element [ dtd ]>` references an internal DTD, one that is embedded within the XML document. The internal DTD must appear between square brackets.

Listing 15-8 presents Listing 15-1 (minus the child elements between the `<recipe>` and `</recipe>` tags) with an internal DTD.

**Listing 15-8. The Recipe Document with an Internal DTD**

```xml
<?xml version="1.0"?>
<!DOCTYPE recipe [
  <!ELEMENT recipe (title, ingredients, instructions)>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT ingredients (ingredient+)>  
  <!ELEMENT ingredient (#PCDATA)>  
  <!ELEMENT instructions (#PCDATA)>  
  <!ATTLIST ingredient qty CDATA "1"> 
]
<recipe>
  <!-- Child elements removed for brevity. -->
</recipe>
```
A document can have internal and external DTDs; for example, `<!DOCTYPE recipe SYSTEM "http://www.tutortutor.ca/dtds/recipe.dtd" [ <!ELEMENT ...]>`. The internal DTD is referred to as the **internal DTD subset** and the external DTD is referred to as the **external DTD subset**. Neither subset can override the element declarations of the other subset.

You can also declare notations and general and parameter entities within DTDs. A **notation** is an arbitrary piece of data that typically describes the format of unparsed binary data, and it typically has the form `<!NOTATION name SYSTEM uri>`, where *name* identifies the notation and *uri* identifies some kind of plug-in that can process the data on behalf of the application that’s parsing the XML document. For example, `<!NOTATION image SYSTEM "psp.exe">` declares a notation named *image* and identifies Windows executable *psp.exe* as a plug-in for processing images.

It’s also common to use notations to specify binary data types via Internet media types (see [http://en.wikipedia.org/wiki/Internet_media_type](http://en.wikipedia.org/wiki/Internet_media_type)). For example, `<!NOTATION image SYSTEM "image/jpeg">` declares an image notation that identifies the *image/jpeg* Internet media type for Joint Photographic Experts Group images.

**General entities** are entities referenced from inside an XML document via **general entity references**—syntactic constructs of the form `&name;`. Examples include the predefined `<`, `>`, `&`, `;`, `amp`, `apos`, and `quot` character entities whose `&lt;`, `&gt;`, `&amp;`, `&apos;`, and `&quot;` character entity references are aliases for characters `<`, `>`, `&`, `;`, ` `, and ` "`, respectively.

General entities are classified as internal or external. An **internal general entity** is a general entity whose value is stored in the DTD and has the form `<!ENTITY name value>`, where *name* identifies the entity and *value* specifies its value. For example, `<!ENTITY copyright "Copyright &copy; 2014 Jeff Friesen. All rights reserved.">` declares an internal general entity named *copyright*. The value of this entity may include another declared entity, such as `&copy;` (the HTML entity for the copyright symbol), and can be referenced from anywhere in an XML document by specifying `&copyright;`.

An **external general entity** is a general entity whose value is stored outside the DTD. The value might be textual data (such as an XML document) or it might be binary data (such as a JPEG image). External general entities are classified as external parsed general entities and external unparsed entities.

An **external parsed general entity** references an external file that stores the entity’s textual data, which is subject to being inserted into a document and parsed by a validating parser when a general entity reference is specified in the document and that has the form `<!ENTITY name SYSTEM uri>`, where *name* identifies the entity and *uri* identifies the external file. For example, `<!ENTITY chapter-header SYSTEM "http://www.tutortutor.ca/entities/chapheader.xml">` identifies chapheader.xml as storing the XML content to be inserted into an XML document wherever `&chapter-header;` appears in the document. The alternative `<!ENTITY name PUBLIC fpi uri>` form can be specified.

**Caution** Because the contents of an external file may be parsed, this content must be well formed.
An external unparsed entity references an external file that stores the entity’s binary data and has the form `<!ENTITY name SYSTEM uri NDATA nname>`, where name identifies the entity, uri locates the external file, and NDATA identifies the notation declaration named nname. The notation typically identifies a plug-in for processing the binary data or the Internet media type of this data. For example, `<!ENTITY photo SYSTEM "photo.jpg" NDATA image>` associates name photo with external binary file photo.png and notation image. The alternative `<!ENTITY name PUBLIC fpi uri NDATA nname>` form can be specified.

Note XML doesn’t allow references to external general entities to appear in attribute values. For example, you cannot specify &chapter-header; in an attribute’s value.

Parameter entities are entities referenced from inside a DTD via parameter entity references, syntactic constructs of the form `%name;`. They're useful for eliminating repetitive content from element declarations. For example, you're creating a DTD for a large company, and this DTD contains three element declarations: `<!ELEMENT salesperson (firstname, lastname)>`, `<!ELEMENT lawyer (firstname, lastname)>`, and `<!ELEMENT accountant (firstname, lastname)>`. Each element contains repeated child element content. If you need to add another child element (such as middleinitial), you'll need to make sure that all of the elements are updated; otherwise, you risk a malformed DTD. Parameter entities can help you solve this problem.

Parameter entities are classified as internal or external. An internal parameter entity is a parameter entity whose value is stored in the DTD and has the form `<!ENTITY % name value>`, where name identifies the entity and value specifies its value. For example, `<!ENTITY % person-name "firstname, lastname">` declares a parameter entity named person-name with value firstname, lastname. Once declared, this entity can be referenced in the three previous element declarations as follows: `<!ELEMENT salesperson (%person-name;)>`, `<!ELEMENT lawyer (%person-name;)>`, and `<!ELEMENT accountant (%person-name;)>`. Instead of adding middleinitial to each of salesperson, lawyer, and accountant, as was done previously, you would now add this child element to person-name, as in `<!ENTITY % person-name "firstname, middleinitial, lastname">`, and this change would be applied to these element declarations.

An external parameter entity is a parameter entity whose value is stored outside the DTD. It has the form `<!ENTITY % name SYSTEM uri>`, where name identifies the entity and uri locates the external file. For example, `<!ENTITY % person-name SYSTEM "http://www.tutortutor.ca/entities/names.dtd">` identifies names.dtd as storing the firstname, lastname text to be inserted into a DTD wherever %person-name; appears in the DTD. The alternative `<!ENTITY % name PUBLIC fpi uri>` form can be specified.

Note This discussion sums up the basics of DTD. One additional topic that wasn't covered (for brevity) is conditional inclusion, which lets you specify those portions of a DTD to make available to parsers and is typically used with parameter entity references.
XML Schema

XML Schema is a grammar language for declaring the structure, content, and semantics (meaning) of an XML document. This language's grammar documents are known as schemas that are themselves XML documents. Schemas must conform to the XML Schema DTD (see www.w3.org/2001/XMLSchema.dtd).

XML Schema was introduced by the W3C to overcome limitations with DTD, such as DTD’s lack of support for namespaces. Also, XML Schema provides an object-oriented approach to declaring an XML document’s grammar. This grammar language provides a much larger set of primitive types than DTD’s CDATA and PCDATA types. For example, you’ll find integer, floating-point, various date and time, and string types to be part of XML Schema.

Note XML Schema predefines 19 primitive types, which are expressed via the following identifiers: anyURI, base64Binary, boolean, date, dateTime, decimal, double, duration, float, hexBinary, gDay, gMonth, gMonthDay, gYear, gYearMonth, NOTATION, QName, string, and time.

XML Schema provides restriction (reducing the set of permitted values through constraints), list (allowing a sequence of values), and union (allowing a choice of values from several types) derivation methods for creating new simple types from these primitive types. For example, XML Schema derives 13 integer types from decimal through restriction; these types are expressed via the following identifiers: byte, int, integer, long, negativeInteger, nonNegativeInteger, positiveInteger, short, unsignedByte, unsignedInt, unsignedLong, and unsignedShort. It also provides support for creating complex types from simple types.

A good way to become familiar with XML Schema is to follow through an example, such as creating a schema for Listing 15-1’s recipe language document. The first step in creating this recipe language schema is to identify all of its elements and attributes. The elements are recipe, title, ingredients, instructions, and ingredient; qty is the solitary attribute.

The next step is to classify the elements according to XML Schema’s content model, which specifies the types of child elements and text nodes (see http://en.wikipedia.org/wiki/Node_(computer_science)) that can be included in an element. An element is considered to be empty when the element has no child elements or text nodes, simple when only text nodes are accepted, complex when only child elements are accepted, and mixed when child elements and text nodes are accepted. None of Listing 15-1’s elements have empty or mixed content models. However, the title, ingredient, and instructions elements have simple content models; and the recipe and ingredients elements have complex content models.

For elements that have a simple content model, you can distinguish between elements having attributes and elements not having attributes. XML Schema classifies elements having a simple content model and no attributes as simple types. Furthermore, it classifies elements having a simple content model and attributes, or elements from other content models as complex types. Furthermore, XML Schema classifies attributes as simple types because they only contain text values; attributes don’t have child elements. Listing 15-1’s title and instructions elements and its qty attribute are simple types. Its recipe, ingredients, and ingredient elements are complex types.
At this point, you can begin to declare the schema. The following example presents the introductory schema element:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

The schema element introduces the grammar. It also assigns the commonly used `xsd` namespace prefix to the standard XML Schema namespace; `xsd:` is subsequently prepended to XML Schema element names.

Next you use the `element` element to declare the `title` and `instructions` simple type elements as follows:

```xml
<xsd:element name="title" type="xs:string"/>
<xsd:element name="instructions" type="xs:string"/>
```

XML Schema requires that each element have a name and (unlike DTD) be associated with a type, which identifies the kind of data stored in the element. For example, the first element declaration identifies `title` as the name via its `name` attribute and `string` as the type via its `type` attribute (string or character data appears between the `<title>` and `</title>` tags). The `xsd:` prefix in `xs:string` is required because `string` is a predefined W3C type.

Continuing, you now use the `attribute` element to declare the `qty` simple type attribute, as follows:

```xml
<xsd:attribute name="qty" type="xs:unsignedInt" default="1"/>
```

This attribute element declares an attribute named `qty`. I’ve chosen `unsignedInt` as this attribute’s type because quantities are nonnegative values. Furthermore, I’ve specified `1` as the default value for when `qty` isn’t specified—attribute elements default to declaring optional attributes.

**Note** The order of element and attribute declarations isn’t significant within a schema.

Now that you’ve declared the simple types, you can start to declare the complex types. To begin, let’s declare `recipe` as follows:

```xml
<xsd:element name="recipe">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="title"/>
         <xsd:element ref="ingredients"/>
         <xsd:element ref="instructions"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

This declaration states that `recipe` is a complex type (via the `complexType` element) consisting of a sequence (via the `sequence` element) of one `title` element followed by one `ingredients` element followed by one `instructions` element. Each of these elements is declared by a different element that’s referred to by its element’s `ref` attribute.
The next complex type to declare is ingredients. The following example provides its declaration:

```xml
<xs:element name="ingredients">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ingredient" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

This declaration states that ingredients is a complex type consisting of a sequence of one or more ingredient elements. The “or more” is specified by including element's maxOccurs attribute and setting this attribute’s value to unbounded.

**Note**  The maxOccurs attribute identifies the maximum number of times that an element can occur. A similar minOccurs attribute identifies the minimum number of times that an element can occur. Each attribute can be assigned 0 or a positive integer. Furthermore, you can specify unbounded for maxOccurs, which means that there’s no upper limit on occurrences of the element. Each attribute defaults to a value of 1, which means that an element can appear only one time when neither attribute is present.

The final complex type to declare is ingredient. Although ingredient can contain only text nodes, which implies that it should be a simple type, it’s the presence of the qty attribute that makes it complex. Check out the following declaration:

```xml
<xs:element name="ingredient">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="qty"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

The element named ingredient is a complex type (because of its optional qty attribute). The simpleContent element indicates that ingredient can only contain simple content (text nodes), and the extension element indicates that ingredient is a new type that extends the predefined string type (specified via the base attribute), implying that ingredient inherits all of string’s attributes and structure. Furthermore, ingredient is given an additional qty attribute.

Listing 15-9 combines the previous examples into a complete schema.
Listing 15-9. The Recipe Document’s Schema

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="title" type="xs:string"/>
  <xs:element name="instructions" type="xs:string"/>
  <xs:attribute name="qty" type="xs:unsignedInt" default="1"/>
  <xs:element name="recipe">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="title"/>
        <xs:element ref="ingredients"/>
        <xs:element ref="instructions"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ingredients">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ingredient" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ingredient">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute ref="qty"/>  
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

After creating the schema, you'll want to reference it from a recipe document. You can accomplish this task by specifying xmlns:xsi and xsi:schemaLocation attributes on the document’s root element start tag (`<recipe>`) as follows:

```xml
<recipe xmlns="http://www.tutortutor.ca/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.tutortutor.ca/schemas recipe.xsd">
```

The `xmlns` attribute identifies `http://www.tutortutor.ca/` as the document’s default namespace. Unprefixed elements and their unprefixed attributes belong to this namespace.

The `xmlns:xsi` attribute associates the conventional xsi (XML Schema Instance) prefix with the standard `http://www.w3.org/2001/XMLSchema-instance` namespace. The only item in the document that's prefixed with xsi: is schemaLocation.

The `schemaLocation` attribute is used to locate the schema. This attribute’s value can be multiple pairs of space-separated values, but it is specified as a single pair of such values in this example. The first value (`http://www.tutortutor.ca/schemas`) identifies the target namespace for the schema, and the second value (`recipe.xsd`) identifies the location of the schema within this namespace.
Note Schema files that conform to XML Schema's grammar are commonly assigned the .xsd file extension.

If an XML document declares a namespace (xmlns default or xmlns:prefix), that namespace must be made available to the schema so that a validating parser can resolve all references to elements and other schema components for that namespace. You also need to mention which namespace the schema describes, and you do so by including the targetNamespace attribute on the schema element. For example, suppose your recipe document declares a default XML namespace, as follows:

```xml
<?xml version="1.0"?>
<recipe xmlns="http://www.tutortutor.ca/">
```

At minimum, you would need to modify Listing 15-9's schema element to include targetNameSpace and the recipe document's default namespace as targetNameSpace's value as follows:

```xml
<xs:schema targetNamespace="http://www.tutortutor.ca/
xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

### Parsing XML Documents with SAX

*Simple API for XML (SAX)* is an event-based API for parsing an XML document sequentially from start to finish. As a SAX-oriented parser encounters an item from the document's *infoset* (an abstract data model describing an XML document's information, see [http://en.wikipedia.org/wiki/XML_Information_Set](http://en.wikipedia.org/wiki/XML_Information_Set)), it makes this item available to an application as an *event* by calling one of the methods in one of the application's *handlers* (an object whose methods are called by the parser to make event information available), which the application has previously registered with the parser. The application can then *consume* this event by processing the infoset item in some manner.

Note According to its official web site ([www.saxproject.org](http://www.saxproject.org)), SAX originated as an XML parsing API for Java. However, SAX isn’t exclusive to Java. Microsoft also supports SAX for its .NET framework (see [http://saxdotnet.sourceforge.net](http://saxdotnet.sourceforge.net)).

After taking you on a tour of the SAX API, I provide a simple demonstration of this API to help you become familiar with its event-based parsing paradigm. I then show you how to create a custom entity resolver.
CHAPTER 15: Parsing, Creating, and Transforming XML Documents

Exploring the SAX API

SAX exists in two major versions. Java implements SAX 1 through the `javax.xml.parsers` package’s abstract `SAXParser` and `SAXParserFactory` classes, and it implements SAX 2 through the `org.xml.sax` package’s `XMLReader` interface and through the `org.xml.sax.helpers` package’s `XMLReaderFactory` class. The `org.xml.sax`, `org.xml.sax.ext`, and `org.xml.sax.helpers` packages provide various types that augment both Java implementations.

**Note** I explore only the SAX 2 implementation because SAX 2 makes available additional infoset items about an XML document (such as comments and CDATA section notifications).

Classes that implement the `XMLReader` interface describe SAX 2-based parsers. Instances of these classes are obtained by calling the `XMLReaderFactory` class’s `createXMLReader()` methods. For example, the following code fragment invokes this class’s `XMLReader createXMLReader()` class method to create and return an `XMLReader` instance:

```java
XMLReader xmlr = XMLReaderFactory.createXMLReader();
```

This method call returns an instance of an `XMLReader`-implementing class and assigns its reference to `xmlr`.

**Note** Behind the scenes, `createXMLReader()` attempts to create an `XMLReader` instance from system defaults according to a lookup procedure that first examines the `org.xml.sax.driver` system property to see if it has a value. If so, this property’s value is used as the name of the class that implements `XMLReader`. Furthermore, an attempt to instantiate this class and return the instance is made. An instance of the `org.xml.sax.SAXException` class is thrown when `createXMLReader()` cannot obtain an appropriate class or instantiate the class.

The returned `XMLReader` object makes available several methods for configuring the parser and parsing a document’s content. These methods are described below:

- `ContentHandler getContentHandler()` returns the current content handler, which is an instance of a class that implements the `org.xml.sax.ContentHandler` interface, or the null reference when none has been registered.
- `DTDHandler getDTDHandler()` returns the current DTD handler, which is an instance of a class that implements the `org.xml.sax.DTDHandler` interface, or the null reference when none has been registered.
- `EntityResolver getEntityResolver()` returns the current entity resolver, which is an instance of a class that implements the `org.xml.sax.EntityResolver` interface, or the null reference when none has been registered.
ErrorHandler getErrorHandler() returns the current error handler, which is an instance of a class that implements the org.xml.sax.ErrorHandler interface, or the null reference when none has been registered.

boolean getFeature(String name) returns the Boolean value that corresponds to the feature identified by name, which must be a fully-qualified URI. This method throws org.xml.sax.SAXNotRecognizedException when the name isn’t recognized as a feature, and it throws org.xml.sax.SAXNotSupportedException when the name is recognized but the associated value cannot be determined when getFeature() is called. SAXNotRecognizedException and SAXNotSupportedException are subclasses of SAXException.

Object getProperty(String name) returns the java.lang.Object instance that corresponds to the property identified by name, which must be a fully-qualified URI. This method throws SAXNotRecognizedException when the name isn’t recognized as a property, and throws SAXNotSupportedException when the name is recognized but the associated value cannot be determined when getProperty() is called.

void parse(InputSource input) parses an XML document and doesn’t return until the document has been parsed. The input parameter stores a reference to an org.xml.sax.InputSource instance, which describes the document’s source (such as a java.io.InputStream instance, or even a java.lang.String-based system identifier URI). This method throws java.io.IOException when the source cannot be read and SAXException when parsing fails, probably due to a well-formedness violation.

void parse(String systemId) parses an XML document by executing parse(new InputSource(systemId));

void setContentHandler(ContentHandler handler) registers the content handler identified by handler with the parser. The ContentHandler interface provides 11 callback methods that are called to report various parsing events (such as the start and end of an element).

void setDTDHandler(DTDHandler handler) registers the DTD handler identified by handler with the parser. The DTDHandler interface provides a pair of callback methods for reporting on notations and external unparsed entities.

void setEntityResolver(EntityResolver resolver) registers the entity resolver identified by resolver with the parser. The EntityResolver interface provides a single callback method for resolving entities.

void setErrorHandler(ErrorHandler handler) registers the error handler identified by handler with the parser. The ErrorHandler interface provides three callback methods that report fatal errors (problems that prevent further parsing, such as well-formedness violations), recoverable errors (problems that don’t prevent further parsing, such as validation failures), and warnings (nonerrors that need to be addressed, such as prefixing an element name with the W3C-reserved xml prefix).
void setFeature(String name, boolean value) assigns value to the feature identified by name, which must be a fully-qualified URI. This method throws SAXNotRecognizedException when the name isn’t recognized as a feature, and it throws SAXNotSupportedException when the name is recognized but the associated value cannot be set when setFeature() is called.

void setProperty(String name, Object value) assigns value to the property identified by name, which must be a fully-qualified URI. This method throws SAXNotRecognizedException when the name isn’t recognized as a property, and it throws SAXNotSupportedException when the name is recognized but the associated value cannot be set when setProperty() is called.

When a handler isn’t installed, all events pertaining to that handler are silently ignored. Not installing an error handler can be problematic because normal processing might not continue and the application wouldn’t be aware that anything had gone wrong. When an entity resolver isn’t installed, the parser performs its own default resolution. I’ll have more to say about entity resolution later in this chapter.

Note You can install a new content handler, DTD handler, entity resolver, or error handler while the document is being parsed. The parser starts using the handler when the next event occurs.

After obtaining an XMLReader instance, you can configure that instance by setting its features and properties. A feature is a name-value pair that describes a parser mode, such as validation. In contrast, a property is a name-value pair that describes some other aspect of the parser interface, such as a lexical handler that augments the content handler by providing callback methods for reporting on comments, CDATA delimiters, and a few other syntactic constructs.

Features and properties have names, which must be absolute URIs beginning with the http:// prefix. A feature’s value is always a Boolean true/false value. In contrast, a property’s value is an arbitrary object. The following example demonstrates setting a feature and a property:

```java
xmlr.setFeature("http://xml.org/sax/features/validation", true);
xmlr.setProperty("http://xml.org/sax/properties/lexical-handler",
    new LexicalHandler() { /* ... */ });
```

The setFeature() call enables the validation feature so that the parser will perform validation. Feature names are prefixed with http://xml.org/sax/features/.
The `setProperty()` call assigns an instance of a class that implements the `org.xml.sax.ext.LexicalHandler` interface to the `lexical-handler` property so that interface methods can be called to report on comments, CDATA sections, and so on. Property names are prefixed with `http://xml.org/sax/properties/`.

**Note**  Parsers must support the namespaces and namespace-prefixes features. namespaces decides whether URLs and local names are passed to `ContentHandler`'s `startElement()` and `endElement()` methods. It defaults to true; these names are passed. The parser can pass empty strings when false. namespace-prefixes decides whether a namespace declaration's `xmlns` and `xmlns:prefix` attributes are included in the Attributes list passed to `startElement()`, and it also decides whether qualified names are passed as the method's third argument; a qualified name is a prefix plus a local name. It defaults to false, meaning that `xmlns` and `xmlns:prefix` aren't included, and meaning that parsers don't have to pass qualified names. No properties are mandatory. The JDK documentation's `org.xml.sax` package page lists standard SAX 2 features and properties.

Features and properties can be read-only or read-write. (In some rare cases, a feature or property might be write-only.) When setting or reading a feature or property, `SAXNotSupportedException` or `SAXNotRecognizedException` might be thrown. For example, if you try to modify a read-only feature/property, an instance of the `SAXNotSupportedException` class is thrown. This exception could also be thrown if you call `setFeature()` or `setProperty()` during parsing. Trying to set the validation feature for a parser that doesn't perform validation is a scenario where an instance of the `SAXNotRecognizedException` class is thrown.

The handlers installed by `setContentHandler()`, `setDTDHandler()`, and `setErrorHandler()`, the entity resolver installed by `setEntityResolver()`, and the handler installed by the `lexical-handler` property/LexicalHandler interface provide various callback methods that you need to understand before you can codify them to respond effectively to parsing events. `ContentHandler` declares the following content-oriented informational callback methods:

- `void characters(char[] ch, int start, int length)` reports an element's character data via the `ch` array. The arguments that are passed to `start` and `length` identify that portion of the array that's relevant to this method call. Characters are passed via a `char[]` array instead of via a String instance as a performance optimization. Parsers commonly store a large amount of the document in an array and repeatedly pass a reference to this array along with updated `start` and `length` values to `characters()`.
void endDocument() reports that the end of the document has been reached. An application might use this method to close an output file or perform some other cleanup.

void endElement(String uri, String localName, String qName) reports that the end of an element has been reached. uri identifies the element's namespace URI, or it is empty when there is no namespace URI or namespace processing hasn't been enabled. localName identifies the element's local name, which is the name without a prefix (the html in html or h:html, for example). qName references the qualified name, for example, h:html or html when there is no prefix.

endElement() is invoked when an end tag is detected, or immediately following startElement() when an empty-element tag is detected.

void endPrefixMapping(String prefix) reports that the end of a namespace prefix mapping (xmlns:h, for example) has been reached, and prefix reports this prefix (h, for example).

void ignorableWhitespace(char[] ch, int start, int length) reports ignorable whitespace (whitespace located between tags where the DTD doesn't allow mixed content). This whitespace is often used to indent tags. The parameters serve the same purpose as those in the characters() method.

void processingInstruction(String target, String data) reports a processing instruction in which target identifies the application to which the instruction is directed and data provides the instruction's data (the null reference when there is no data).

void setDocumentLocator(Locator locator) reports an org.xml.sax.Locator object (an instance of a class implementing the Locator interface) whose int getColumnNumber(), int getLineNumber(), String getPublicId(), and String getSystemId() methods can be called to obtain location information at the end position of any document-related event, even when the parser isn’t reporting an error. This method is called before startDocument(), and it is a good place to save the Locator object so that it can be accessed from other callback methods.

void skippedEntity(String name) reports all skipped entities. Validating parsers resolve all general entity references, but nonvalidating parsers have the option of skipping them because nonvalidating parsers don’t read DTDs where these entities are declared. If the nonvalidating parser doesn’t read a DTD, it will not know if an entity is properly declared. Instead of attempting to read the DTD and report the entity's replacement text, the nonvalidating parser calls skippedEntity() with the entity's name.

void startDocument() reports that the start of the document has been reached. An application might use this method to create an output file or perform some other initialization.

void startElement(String uri, String localName, String qName, Attributes attributes) reports that the start of an element has been reached. uri identifies the element's namespace URI or is empty when there is no namespace URI or namespace processing hasn’t been enabled. localName identifies the element’s local name, qName references its qualified name, and attributes references an
array of org.xml.sax.Attribute objects that identify the element's attributes; this array is empty when there are no attributes. `startElement()` is invoked when a start tag or an empty-element tag is detected.

- `void startPrefixMapping(String prefix, String uri)` reports that the start of a namespace prefix mapping (xmlns:h="http://www.w3.org/1999/xhtml", for example) has been reached in which `prefix` reports this prefix (such as h) and `uri` reports the URI to which the prefix is mapped (http://www.w3.org/1999/xhtml, for example).

Each method except for `setDocumentLocator()` is declared to throw `SAXException`, which an overriding callback method might choose to throw when it detects a problem.

`DTDHandler` declares the following DTD-oriented informational callback methods:

- `void notationDecl(String name, String publicId, String systemId)` reports a notation declaration, in which `name` provides this declaration's name attribute value, `publicId` provides this declaration's public attribute value (the null reference when this value isn't available), and `systemId` provides this declaration's system attribute value.

- `void unparsedEntityDecl(String name, String publicId, String systemId, String notationName)` reports an external unparsed entity declaration in which `name` provides the value of this declaration's name attribute, `publicId` provides the value of the public attribute (the null reference when this value isn't available), `systemId` provides the value of the system attribute, and `notationName` provides the NDATA name.

Each method is declared to throw `SAXException`, which an overriding callback method might choose to throw when it detects a problem.

`EntityResolver` declares the following callback method:

- `InputSource resolveEntity(String publicId, String systemId)` is called to let the application resolve an external entity (such as an external DTD subset) by returning a custom `InputSource` instance that's based on a different URI. This method is declared to throw `SAXException` when it detects a SAX-oriented problem, and it is also declared to throw `IOException` when it encounters an I/O error, possibly in response to creating an `InputStream` instance or a `java.io.Reader` instance for the `InputSource` being created.

`ErrorHandler` declares the following error-oriented informational callback methods:

- `void error(SAXParseException exception)` reports that a recoverable parser error (typically the document isn't valid) has occurred; the details are specified via the argument passed to `exception`. This method is typically overridden to report the error via a command window or to log it to a file or a database.

- `void fatalError(SAXParseException exception)` reports that an unrecoverable parser error (the document isn't well formed) has occurred; the details are specified via the argument passed to `exception`. This method is typically overridden so that the application can log the error before it stops processing the document (because the document is no longer reliable).
void warning(SAXParseException e) reports that a nonserious error (such as an element name beginning with the reserved xml character sequence) has occurred; the details are specified via the argument passed to exception. This method is typically overridden to report the warning via a console or to log it to a file or a database.

Each method is declared to throw SAXException, which an overriding callback method might choose to throw when it detects a problem.

LexicalHandler declares the following additional content-oriented informational callback methods:

- void comment(char[] ch, int start, int length) reports a comment via the ch array. The arguments that are passed to start and length identify that portion of the array that’s relevant to this method call.
- void endCDATA() reports the end of a CDATA section.
- void endDTD() reports the end of a DTD.
- void endEntity(String name) reports the start of the entity identified by name.
- void startCDATA() reports the start of a CDATA section.
- void startDTD(String name, String publicId, String systemId) reports the start of the DTD identified by name. publicId specifies the declared public identifier for the external DTD subset or is the null reference when none was declared. Similarly, systemId specifies the declared system identifier for the external DTD subset or is the null reference when none was declared.
- void startEntity(String name) reports the start of the entity identified by name.

Each method is declared to throw SAXException, which an overriding callback method might choose to throw when it detects a problem.

Because it can be tedious to implement all of the methods in each interface, the SAX API conveniently provides the org.xml.sax.helpers.DefaultHandler adapter class to relieve you of this tedium. DefaultHandler implements ContentHandler, DTDHandler, EntityResolver, and ErrorHandler. SAX also provides org.xml.sax.ext.DefaultHandler2, which subclasses DefaultHandler and which also implements LexicalHandler.

**Demonstrating the SAX API**

Listing 15-10 presents the source code to SAXDemo, an application that demonstrates the SAX API. The application consists of a SAXDemo entry-point class and a Handler subclass of DefaultHandler2.

Listing 15-10. SAXDemo

```java
import java.io.FileReader;
import java.io.IOException;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
```
import org.xml.sax.helpers.XMLReaderFactory;

public class SAXDemo {
    public static void main(String[] args) {
        if (args.length < 1 || args.length > 2) {
            System.err.println("usage: java SAXDemo xmlfile [v]"COVERED PAGE
                    return;
        }
        try {
            XMLReader xmlr = XMLReaderFactory.createXMLReader();
            if (args.length == 2 && args[1].equals("v")) {
                xmlr.setFeature("http://xml.org/sax/features/validation", true);
                xmlr.setFeature("http://xml.org/sax/features/namespace-prefixes", true);
            }
            Handler handler = new Handler();
            xmlr.setContentHandler(handler);
            xmlr.setDTDHandler(handler);
            xmlr.setEntityResolver(handler);
            xmlr.setErrorHandler(handler);
            xmlr.setProperty("http://xml.org/sax/properties/lexical-handler", handler);
            xmlr.parse(new InputSource(new FileReader(args[0])));
        } catch (IOException ioe) {
            System.err.println("IOE: " + ioe);
        } catch (SAXException saxe) {
            System.err.println("SAXE: " + saxe);
        }
    }
}

SAXDemo's main() method first verifies that one or two command-line arguments (the name of an XML document optionally followed by lowercase letter v, which tells SAXDemo to create a validating parser) have been specified. It then creates an XMLReader instance; conditionally enables the validation feature and enables the namespace-prefixes feature; instantiates the companion Handler class; installs this Handler instance as the parser's content handler, DTD handler, entity resolver, and error handler; installs this Handler instance as the value of the lexical-handler property; creates an input source to read the document from a file; and parses the document.

The Handler class's source code is presented in Listing 15-11.
Listing 15-11. Handler

```java
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.Locator;
import org.xml.sax.SAXParseException;
import org.xml.sax.ext.DefaultHandler2;

public class Handler extends DefaultHandler2 {
    private Locator locator;

    @Override
    public void characters(char[] ch, int start, int length) {
        System.out.print("characters() [");
        for (int i = start; i < start + length; i++)
            System.out.print(ch[i]);
        System.out.println("]");
    }

    @Override
    public void comment(char[] ch, int start, int length) {
        System.out.print("characters() [");
        for (int i = start; i < start + length; i++)
            System.out.print(ch[i]);
        System.out.println("]");
    }

    @Override
    public void endCDATA() {
        System.out.println("endCDATA()");
    }

    @Override
    public void endDocument() {
        System.out.println("endDocument()");
    }

    @Override
    public void endDTD() {
        System.out.println("endDTD()");
    }
}```
@Override
public void endElement(String uri, String localName, String qName)
{
    System.out.print("endElement() ");
    System.out.print("uri=[" + uri + ", ");
    System.out.print("localName=[" + localName + ", ");
    System.out.println("qName=[" + qName + "]");
}

@Override
public void endEntity(String name)
{
    System.out.print("endEntity() ");
    System.out.println("name=[" + name + "]");
}

@Override
public void endPrefixMapping(String prefix)
{
    System.out.print("endPrefixMapping() ");
    System.out.println("prefix=[" + prefix + "]");
}

@Override
public void error(SAXParseException saxpe)
{
    System.out.println("error() "+ saxpe);
}

@Override
public void fatalError(SAXParseException saxpe)
{
    System.out.println("fatalError() "+ saxpe);
}

@Override
public void ignorableWhitespace(char[] ch, int start, int length)
{
    System.out.print("ignorableWhitespace() [");
    for (int i = start; i < start + length; i++)
        System.out.print(ch[i]);
    System.out.println("]");
}

@Override
public void notationDecl(String name, String publicId, String systemId)
{
    System.out.print("notationDecl() ");
    System.out.print("name=[" + name + "]");
    System.out.print("publicId=[" + publicId + "]");
    System.out.println("systemId=[" + systemId + "]");
}
@Override
public void processingInstruction(String target, String data)
{
    System.out.print("processingInstruction() [";
    System.out.println("target=[" + target + "]");
    System.out.println("data=[" + data + "]");
}

@Override
public InputSource resolveEntity(String publicId, String systemId)
{
    System.out.print("resolveEntity() ");
    System.out.print("publicId=[" + publicId + "]");
    System.out.println("systemId=[" + systemId + "]");
    // Do not perform a remapping.
    InputSource is = new InputSource();
    is.setPublicId(publicId);
    is.setSystemId(systemId);
    return is;
}

@Override
public void setDocumentLocator(Locator locator)
{
    System.out.print("setDocumentLocator() ");
    System.out.println("locator=[" + locator + "]");
    this.locator = locator;
}

@Override
public void skippedEntity(String name)
{
    System.out.print("skippedEntity() ");
    System.out.println("name=[" + name + "]");
}

@Override
public void startCDATA()
{
    System.out.println("startCDATA()");
}

@Override
public void startDocument()
{
    System.out.println("startDocument()");
}

@Override
public void startDTD(String name, String publicId, String systemId)
{
    System.out.print("startDTD() ");
}
@Override
public void startElement(String uri, String localName, String qName,
Attributes attributes)
{
    System.out.print("startElement() ");
    System.out.print("uri=\" + uri + \"], ");
    System.out.print("localName=\" + localName + \"], ");
    System.out.println("qName=\" + qName + \"]");
    for (int i = 0; i < attributes.getLength(); i++)
        System.out.println("  Attribute: " + attributes.getLocalName(i) + ", " +
            attributes.getValue(i));
    System.out.println("Column number=\" + locator.getColumnNumber() + \"]");
    System.out.println("Line number=\" + locator.getLineNumber() + \"]");
}

@Override
public void startEntity(String name)
{
    System.out.print("startEntity() ");
    System.out.println("name=\" + name + \"]");
}

@Override
public void startPrefixMapping(String prefix, String uri)
{
    System.out.print("startPrefixMapping() ");
    System.out.print("prefix=\" + prefix + \"]; ");
    System.out.println("uri=\" + uri + \"]");
}

@Override
public void unparsedEntityDecl(String name, String publicId,
String systemId, String notationName)
{
    System.out.print("unparsedEntityDecl() ");
    System.out.print("name=\" + name + \"]");
    System.out.print("publicId=\" + publicId + \"]");
    System.out.print("systemId=\" + systemId + \"]");
    System.out.println("notationName=\" + notationName + \"]");
}

@Override
public void warning(SAXParseException saxpe)
{
    System.out.println("warning() " + saxpe);
}
The Handler subclass is pretty straightforward; it outputs every possible piece of information about an XML document, subject to feature and property settings. You’ll find this class handy for exploring the order in which events occur along with various features and properties.

After compiling SAXDemo.java and Handler.java (javac SAXDemo.java), execute the following command to parse Listing 15-4’s svg-examples.xml document:

```java
java SAXDemo svg-examples.xml
```

SAXDemo responds by presenting the following output (the hash code may be different):

```java
setDocumentLocator() locator=[com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$LocatorProxy@1395ddba]
startDocument()
startElement() uri=[], localName=[svg-examples], qName=[svg-examples]
   Column number=[15]
   Line number=[2]
   characters() [
   ]
startElement() uri=[], localName=[example], qName=[example]
   Column number=[13]
   Line number=[3]
   characters() [
      The following Scalable Vector Graphics document describes a blue-filled and ]
   characters() [
      black-stroked rectangle.
   ]
startCDATA()
caracters() [<svg width="100%" height="100%" version="1.1"
   xmlns="http://www.w3.org/2000/svg">
   <rect width="300" height="100"
      style="fill:rgb(0,0,255);stroke-width:1; stroke:rgb(0,0,0)"/>
</svg>]
endCDATA()
caracters() [
endElement() uri=[], localName=[example], qName=[example]
characters() [
endElement() uri=[], localName=[svg-examples], qName=[svg-examples]
endDocument()
```

The first output line proves that setDocumentLocator() is called first. It also identifies the Locator instance whose getColumnNumber() and getLineNumber() methods are called to output the parser location when startElement() is called; these methods return column and line numbers starting at 1.

Perhaps you’re curious about the three instances of the following output:

```java
characters() [
]
The instance of this output that follows the `endCDATA()` output is reporting a carriage return/line feed combination that wasn’t included in the preceding `characters()` method call, which was passed the contents of the CDATA section minus these line terminator characters. In contrast, the instances of this output that follow the `startElement()` call for `svg-examples` and follow the `endElement()` call for `example` are somewhat curious. There’s no content between `<svg-examples>` and `<example>`, and between `</example>` and `</svg-examples>`, or is there?

You can satisfy this curiosity by modifying `svg-examples.xml` to include an internal DTD. Place the following DTD (which indicates that an `svg-examples` element contains one or more `example` elements, and that an `example` element contains parsed character data) between the XML declaration and the `<svg-examples>` start tag:

```xml
<!DOCTYPE svg-examples [
  <!ELEMENT svg-examples (example+)>
  <!ELEMENT example (#PCDATA)>
]>
```

Continuing, execute the following command:

```bash
java SAXDemo svg-examples.xml
```

This time, you should see the following output (although the hash code will probably differ):

```xml
setDocumentLocator() locator=[com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$LocatorProxy@540fe861]
startDocument()
startDTD() name=[svg-examples]publicId=[null]systemId=[null]
endDTD()
startElement() uri=[], localName=[svg-examples], qName=[svg-examples]
  Column number=[15]
  Line number=[6]
  ignorableWhitespace()
startElement() uri=[], localName=[example], qName=[example]
  Column number=[13]
  Line number=[7]
  characters()
    The following Scalable Vector Graphics document describes a blue-filled and black-stroked rectangle.
  characters()
startCDATA()
characters() [<svg width="100%" height="100%" version="1.1"
  xmlns="http://www.w3.org/2000/svg">
  <rect width="300" height="100"
    style="fill:rgb(0,0,255);stroke-width:1; stroke:rgb(0,0,0")/>
</svg>
endCDATA()
characters()
]
endElement() uri=[], localName=[example], qName=[example]
ignorableWhitespace() [
]
endElement() uri=[], localName=[svg-examples], qName=[svg-examples]
endDocument()

This output reveals that the ignorableWhitespace() method was called after startElement() for svg-examples and after endElement() for example. The former two calls to characters() that produced the strange output were reporting ignorable whitespace.

Recall that I previously defined ignorable whitespace as whitespace located between tags where the DTD doesn’t allow mixed content. For example, the DTD indicates that svg-examples shall contain only example elements, not example elements and parsed character data. However, the line terminator following the <svg-examples> tag and the leading whitespace before <example> are parsed character data. The parser now reports these characters by calling ignorableWhitespace().

This time, there are only two occurrences of the following output:

characters() [
      
]  

The first occurrence reports the line terminator separately from the example element’s text (before the CDATA section); it didn’t do so previously, which proves that characters() is called with either all or part of an element’s content. Once again, the second occurrence reports the line terminator that follows the CDATA section.

Let’s validate svg-examples.xml without the previously presented internal DTD. You’ll do so by executing the following command; don’t forget to include the v command-line argument or the document won’t validate:

java SAXDemo svg-examples.xml v

Among its output are a couple of error()-prefixed lines that are similar to those shown below:

error() org.xml.sax.SAXParseException; lineNumber: 2; columnNumber: 14; Document is invalid: no grammar found.
error() org.xml.sax.SAXParseException; lineNumber: 2; columnNumber: 14; Document root element "svg-examples", must match DOCTYPE root "null".

These lines reveal that a DTD grammar hasn’t been found. Furthermore, the parser reports a mismatch between svg-examples (it considers the first encountered element to be the root element) and null (it considers null to be the name of the root element in the absence of a DTD). Neither violation is considered to be fatal, which is why error() is called instead of fatalError().

Add the internal DTD to svg-examples.xml and re-execute java SAXDemo svg-examples.xml v. This time, you should see no error()-prefixed lines in the output.
Tip  SAX 2 validation defaults to validating against a DTD. To validate against an XML Schema-based schema instead, add the schemaLanguage property with the http://www.w3.org/2001/XMLSchema value to the XMLReader instance. Accomplish this task for SAXDemo by specifying xmlr.setProperty("http://java.sun.com/xml/jaxp/properties/schemaLanguage", "http://www.w3.org/2001/XMLSchema"); before xmlr.parse(new InputSource(new FileReader(args[0])));

Creating a Custom Entity Resolver

While exploring XML, I introduced you to the concept of entities, which are aliased data. Then I discussed general entities and parameter entities in terms of their internal and external variants.

Unlike internal entities, whose values are specified in a DTD, the values of external entities are specified outside of a DTD and are identified via public and/or system identifiers. The system identifier is a URI, whereas the public identifier is a formal public identifier.

An XML parser reads an external entity (including the external DTD subset) via an InputSource instance that’s connected to the appropriate system identifier. In many cases, you pass a system identifier or InputSource instance to the parser and let it discover where to find other entities that are referenced from the current document entity.

However, for performance or other reasons, you might want the parser to read the external entity’s value from a different system identifier, such as a local DTD copy’s system identifier. You can accomplish this task by creating an entity resolver that uses the public identifier to choose a different system identifier. Upon encountering an external entity, the parser calls the custom entity resolver to obtain this identifier.

Consider Listing 15-12’s formal specification of Listing 15-1’s grilled cheese sandwich recipe.

Listing 15-12. XML-Based Recipe for a Grilled Cheese Sandwich Specified in Recipe Markup Language

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recipeml PUBLIC "-//FormatData//DTD RecipeML 0.5//EN"
  "http://www.formatdata.com/recipeml/recipeml.dtd">
<recipeml version="0.5">
  <recipe>
    <head>
      <title>Grilled Cheese Sandwich</title>
    </head>
    <ingredients>
      <ing>
        <amt><qty>2</qty><unit>slice</unit></amt>
        <item>bread</item>
      </ing>
      <ing>
        <amt><qty>1</qty><unit>slice</unit></amt>
        <item>cheese</item>
      </ing>
    </ingredients>
  </recipe>
</recipeml>
```
Listing 15-12 specifies the grilled cheese sandwich recipe in *Recipe Markup Language (RecipeML)*, an XML-based language for marking up recipes. (A company named FormatData [see www.formatdata.com](http://www.formatdata.com) released this format in 2000.)

The document type declaration reports `-//FormatData//DTD RecipeML 0.5//EN` as the formal public identifier and `http://www.formatdata.com/recipeml/recipeml.dtd` as the system identifier. Instead of keeping the default mapping, let’s map this formal public identifier to `recipeml.dtd`, a system identifier for a local copy of this DTD file.

To create a custom entity resolver to perform this mapping, you declare a class that implements the `EntityResolver` interface in terms of its `InputSource resolveEntity(String publicId, String systemId)` method. You then use the passed `publicId` value as a key into a map that points to the desired `systemId` value, and then use this value to create and return a custom `InputSource`. Listing 15-13 presents the resulting class.

**Listing 15-13. LocalRecipeML**

```java
import java.util.HashMap;
import java.util.Map;
import org.xml.sax.EntityResolver;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

public class LocalRecipeML implements EntityResolver
{
  private Map<String, String> mappings = new HashMap<String, String>();

  LocalRecipeML()
  {
    mappings.put("-//FormatData//DTD RecipeML 0.5//EN", "recipeml.dtd");
  }
```

```
public InputSource resolveEntity(String publicId, String systemId) {
    if (mappings.containsKey(publicId)) {
        System.out.println("obtaining cached recipeml.dtd");
        systemId = mappings.get(publicId);
        InputSource localSource = new InputSource(systemId);
        return localSource;
    }
    return null;
}

Listing 15-13 declares LocalRecipeML. This class's constructor stores the formal public identifier for the RecipeML DTD and the system identifier for a local copy of this DTD's document in a map.

Note Although it's unnecessary to use a map in this example (an if (publicId.equals("-//FormatData//DTD RecipeML 0.5//EN")) return new InputSource("recipeml.dtd") else return null; statement would suffice), I've chosen to use a map in case I want to expand the number of mappings in the future. In another scenario, you would probably find a map to be very convenient. For example, it's easier to use a map than to use a series of if statements in a custom entity resolver that maps XHTML's strict, transitional, and frameset formal public identifiers, and that also maps its various entity sets to local copies of these document files.

The overriding resolveEntity() method uses publicId's argument to locate the corresponding system identifier in the map; the systemId parameter value is ignored because it never refers to the local copy of recipeml.dtd. When the mapping is found, an InputSource object is created and returned. If the mapping couldn't be found, the null reference would be returned.

To install this custom entity resolver in SAXDemo, specify xmlr.setEntityResolver(new LocalRecipeML()); before the parse() method call. After recompiling the source code, execute the following command:

demo SAXDemo gcs.xml

Here, gcs.xml stores Listing 15-12's text. In the resulting output, you should observe the message "obtaining cached recipeml.dtd" before the call to startEntity().

Tip The SAX API includes an org.xml.sax.ext.EntityResolver2 interface that provides improved support for resolving entities. If you prefer to implement EntityResolver2 instead of EntityResolver, replace the setEntityResolver() call to install the entity resolver with a setFeature() call whose feature name is use-entity-resolver2 (don't forget the http://xml.org/sax/features/ prefix).
## Parsing and Creating XML Documents with DOM

*Document Object Model (DOM)* is an API for parsing an XML document into an in-memory tree of nodes and for creating an XML document from a tree of nodes. After a DOM parser has created a document tree, an application uses the DOM API to navigate over and extract infoset items from the tree’s nodes.

**Note** DOM originated as an object model for the Netscape Navigator 3 and Microsoft Internet Explorer 3 web browsers. Collectively, these implementations are known as DOM Level 0. Because each vendor’s DOM implementation was only slightly compatible with the other, the W3C subsequently took charge of DOM’s development to promote standardization, and has so far released DOM Levels 1, 2, and 3 (with Level 4 under development). Java 7 and newer versions of Android support all three DOM levels through their DOM APIs.

DOM has two big advantages over SAX. First, DOM permits random access to a document’s infoset items whereas SAX only permits serial access. Second, DOM lets you also create XML documents whereas you can only parse documents with SAX. However, SAX is advantageous over DOM in that it can parse documents of arbitrary size, whereas the size of documents parsed or created by DOM is limited by the amount of available memory for storing the document’s node-based tree structure.

In this section, I first introduce you to DOM’s tree structure. I then take you on a tour of the DOM API; you learn how to use this API to parse and create XML documents.

### A Tree of Nodes

DOM views an XML document as a tree that’s composed of several kinds of nodes. This tree has a single root node, and all nodes except for the root have a parent node. Also, each node has a list of child nodes. When this list is empty, the child node is known as a *leaf node*.

**Note** DOM permits nodes to exist that are not part of the tree structure. For example, an element node’s attribute nodes are not regarded as child nodes of the element node. Also, nodes can be created but not inserted into the tree; they can also be removed from the tree.

Each node has a *node name*, which is the complete name for nodes that have names (such as an element’s or an attribute’s prefixed name), and *node-type* for unnamed nodes, where node-type is one of cdata-section, comment, document, document-fragment, or text. Nodes also have *local names* (names without prefixes), prefixes, and namespace URIs (although these attributes may be null for certain kinds of nodes, such as comments). Finally, nodes have string values, which happen to be the content of text nodes, comment nodes, and similar text-oriented nodes; normalized values of attributes; and null for everything else.
DOM classifies nodes into 12 types, of which seven types can be considered part of a DOM tree. All of these types are described below:

- **Attribute node**: One of an element’s attributes. It has a name, a local name, a prefix, a namespace URI, and a normalized string value. The value is normalized by resolving any entity references and by converting sequences of whitespace to a single whitespace character. An attribute node has children, which are the text and any entity reference nodes that form its value. Attributes nodes are not regarded as children of their associated element nodes.

- **CDATA section node**: The contents of a CDATA section. Its name is `#cdata-section`, and its value is the CDATA section’s text.

- **Comment node**: A document comment. Its name is `#comment`, and its value is the comment text. A comment node has a parent, which is the node that contains the comment.

- **Document node**: The root of a DOM tree. Its name is `#document`, it always has a single element node child, and it will also have a document type child node when the document has a document type declaration. Furthermore, it can have additional child nodes describing comments or processing instructions that appear before or after the root element’s start tag. There can be only one document node in the tree.

- **Document fragment node**: An alternative root node. Its name is `#document-fragment`, and it contains anything that an element node can contain (such as other element nodes and even comment nodes). A parser never creates this kind of a node. However, an application can create a document fragment node when it extracts part of a DOM tree to be moved somewhere else. Document fragment nodes let you work with subtrees.

- **Document type node**: A document type declaration. Its name is the name specified by the document type declaration for the root element. Also, it has a (possibly null) public identifier, a required system identifier, an internal DTD subset (which is possibly null), a parent (the document node that contains the document type node), and lists of DTD-declared notations and general entities. Its value is always set to null.

- **Element node**: A document’s element. It has a name, a local name, a (possibly null) prefix, and a namespace URI, which is null when the element doesn’t belong to any namespace. An element node contains children, including text nodes, and even comment and processing instruction nodes.

- **Entity node**: The parsed and unparsed entities that are declared in a document’s DTD. When a parser reads a DTD, it attaches a map of entity nodes (indexed by entity name) to the document type node. An entity node has a name and a system identifier, and it can also have a public identifier if one appears in the DTD. Finally, when the parser reads the entity, the entity node is given a list of read-only child nodes that contain the entity’s replacement text.
Entity reference node: A reference to a DTD-declared entity. Each entity reference node has a name, and it is included in the tree when the parser doesn’t replace entity references with their values. The parser never includes entity reference nodes for character references (such as &amp; or &#{039;}) because they’re replaced by their respective characters and included in a text node.

Notation node: A DTD-declared notation. A parser that reads the DTD attaches a map of notation nodes (indexed by notation name) to the document type node. Each notation node has a name and a public identifier or a system identifier, whichever identifier was used to declare the notation in the DTD. Notation nodes don’t have children.

Processing instruction node: A processing instruction that appears in the document. It has a name (the instruction’s target), a string value (the instruction’s data), and a parent (its containing node).

Text node: Document content. Its name is #text, and it represents a portion of an element’s content when an intervening node (such as a comment) must be created. Characters such as < and & that are represented in the document via character references are replaced by the literal characters they represent. When these nodes are written to a document, these characters must be escaped.

Although these node types store considerable information about an XML document, there are limitations (such as not exposing whitespace outside of the root element). In contrast, most DTD or schema information, such as element types (<!ELEMENT...>) and attribute types (<xs:attribute...>), cannot be accessed through the DOM.

DOM Level 3 addresses some of the DOM’s various limitations. For example, although DOM doesn’t provide a node type for the XML declaration, DOM Level 3 makes it possible to access the XML declaration’s version, encoding, and standalone attribute values via attributes of the document node.

Nonroot nodes never exist in isolation. For example, it’s never the case for an element node not to belong to a document or to a document fragment. Even when such nodes are disconnected from the main tree, they remain aware of the document or document fragment to which they belong.

Exploring the DOM API

Java implements DOM through the javax.xml.parsers package’s abstract DocumentBuilder and DocumentBuilderFactory classes along with the nonabstract FactoryConfigurationException and ParserConfigurationException classes. The org.w3c.dom, org.w3c.dom.bootstrap (not supported by Android), org.w3c.dom.events (not supported by Android), and org.w3c.dom.ls packages provide various types that augment this implementation.

The first step in working with DOM is to instantiate DocumentBuilderFactory by calling one of its newInstance() methods. For example, the following code fragment invokes DocumentBuilderFactory’s DocumentBuilderFactory.newInstance() class method:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```
Behind the scenes, newInstance() follows an ordered lookup procedure to identify the DocumentBuilderFactory implementation class to load. This procedure first examines the javax.xml.parsers.DocumentBuilderFactory system property and lastly chooses the Java platform's default DocumentBuilderFactory implementation class when no other class is found. If an implementation class isn't available (perhaps the class identified by the javax.xml.parsers.DocumentBuilderFactory system property doesn't exist) or cannot be instantiated, newInstance() throws an instance of the FactoryConfigurationError class. Otherwise, it instantiates the class and returns its instance.

After obtaining a DocumentBuilderFactory instance, you can call various configuration methods to configure the factory. For example, you could call DocumentBuilderFactory's void setNamespaceAware(boolean awareness) method with a true argument to tell the factory that any returned parser (known as a document builder to DOM) must provide support for XML namespaces. You can also call void setValidating(boolean validating) with true as the argument to validate documents against their DTDs, or call void setSchema(Schema schema) to validate documents against the javax.xml.validation.Schema instance identified by schema.

## VALIDATION API

Schema is a member of the Validation API, which decouples document parsing from validation, making it easier for applications to take advantage of specialized validation libraries that support additional schema languages (such as Relax NG—see [http://en.wikipedia.org/wiki/RELAX_NG](http://en.wikipedia.org/wiki/RELAX_NG)), and also making it easier to specify the location of a schema.

The Validation API is associated with the javax.xml.validation package, which also includes SchemaFactory, SchemaFactoryLoader, TypeInfoProvider, Validator, and ValidatorHandler. Schema is the central class, and it represents an immutable in-memory representation of a grammar.

DOM supports the Validation API via DocumentBuilderFactory's void setSchema(Schema schema) and Schema getSchema() methods. Similarly, SAX 1.0 supports Validation via SAXParserFactory's void setSchema(Schema schema) and Schema getSchema() methods. SAX 2.0 and StAX don't support the Validation API.

The following example provides a demonstration of the Validation API in a DOM context:

```java
// Parse an XML document into a DOM tree.
Document document = parser.parse(new File("instance.xml"));
// Create a SchemaFactory capable of understanding W3C XML Schema (WXS).
SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
// Load a WXS schema, represented by a Schema instance.
Source schemaFile = new StreamSource(new File("mySchema.xsd"));
Schema schema = factory.newSchema(schemaFile);
// Create a Validator instance, which is used to validate an XML document.
Validator validator = schema.newValidator();
// Validate the DOM tree.
try
```
CHAPTER 15: Parsing, Creating, and Transforming XML Documents

```java
{   validator.validate(new DOMSource(document));
}
catch (SAXException saxe)
{
    // XML document is invalid!
}
```

This example refers to XSLT types such as Source. I explore XSLT later in this chapter.

After the factory has been configured, call its DocumentBuilder newDocumentBuilder() method to return a document builder that supports the configuration, as demonstrated here:

```java
DocumentBuilder db = dbf.newDocumentBuilder();
```

If a document builder cannot be returned (perhaps the factory cannot create a document builder that supports XML namespaces), this method throws a ParserConfigurationException instance.

Assuming that you’ve successfully obtained a document builder, what happens next depends upon whether you want to parse or create an XML document.

**Parsing XML Documents**

DocumentBuilder provides several overloaded parse() methods for parsing an XML document into a node tree. These methods differ in how they obtain the document. For example, Document parse(String uri) parses the document that’s identified by its string-based URI argument.

> **Note** Each parse() method throws java.lang.IllegalArgumentException when null is passed as the method’s first argument, IOException when an input/output problem occurs, and SAXException when the document cannot be parsed. This last exception type implies that DocumentBuilder’s parse() methods rely on SAX to take care of the actual parsing work. Because they are more involved in building the node tree, DOM parsers are commonly referred to as document builders.

The returned org.w3c.dom.Document object provides access to the parsed document through methods such as DocumentType getDoctype(), which makes the document type declaration available through the org.w3c.dom.DocumentType interface. Conceptually, Document is the root of the document’s node tree.

> **Note** Apart from DocumentBuilder, DocumentBuilderFactory, and a few other classes, DOM is based on interfaces, of which Document and DocumentType are examples. Behind the scenes, DOM methods (such as the parse() methods) return objects whose classes implement these interfaces.
Document and all other \texttt{org.w3c.dom} interfaces that describe different kinds of nodes are subinterfaces of the \texttt{org.w3c.dom.Node} interface. As such, they inherit Node's constants and methods.

Node declares 12 constants that represent the various kinds of nodes; \texttt{ATTRIBUTE\_NODE} and \texttt{ELEMENT\_NODE} are examples. When you want to identify the kind of node represented by a given Node object, call Node's short \texttt{getNodeType()} method and compare the returned value to one of these constants.

\textbf{Note} The rationale for using \texttt{getNodeType()} and these constants, instead of using \texttt{instanceof} and a class name, is that DOM (the object model, not the Java DOM API) was designed to be language independent, and languages such as AppleScript don't have the equivalent of \texttt{instanceof}.

Node declares several methods for getting and setting common node properties. These methods include \texttt{String getNodeName()}, \texttt{String getLocalName()}, \texttt{String getNamespaceURI()}, \texttt{String getPrefix()}, \texttt{void setPrefix(String prefix)}, \texttt{String getNodeValue()}, and \texttt{void setNodeValue(String nodeValue)}, which let you get and (for some properties) set a node's name (such as \texttt{#text}), local name, namespace URI, prefix, and normalized string value properties.

\textbf{Note} Various Node methods (such as \texttt{setPrefix()} and \texttt{getNodeValue()}) throw an instance of the \texttt{org.w3c.dom.DOMException} class when something goes wrong. For example, \texttt{setPrefix()} throws this exception when the prefix argument contains an illegal character, the node is read-only, or the argument is malformed. Similarly, \texttt{getNodeValue()} throws \texttt{DOMException} when \texttt{getNodeValue()} would return more characters than can fit into a \texttt{DOMString} (a W3C type) variable on the implementation platform. \texttt{DOMException} declares a series of constants (such as \texttt{DOMSTRING\_SIZE\_ERR}) that classify the reason for the exception.

Node declares several methods for navigating the node tree. Three of its navigation methods are as follows:

\begin{itemize}
  \item \texttt{boolean hasChildNodes()} returns true when a node has child nodes.
  \item \texttt{Node getFirstChild()} returns the node's first child.
  \item \texttt{Node getLastChild()} returns the node's last child.
\end{itemize}

For nodes with multiple children, you'll find the \texttt{NodeList} \texttt{getChildNodes()} method to be handy. This method returns an \texttt{org.w3c.dom.NodeList} instance whose \texttt{int getLength()} method returns the number of nodes in the list and whose \texttt{Node item(int index)} method returns the node at the index\texttt{th} position in the list (or null when index's value isn't valid; it's less than 0 or greater than or equal to \texttt{getLength()}'s value).
Node declares four methods for modifying the tree by inserting, removing, replacing, and appending child nodes:

- **Node insertBefore(Node newChild, Node refChild)** inserts `newChild` before the existing node specified by `refChild` and returns `newChild`.
- **Node removeChild(Node oldChild)** removes the child node identified by `oldChild` from the tree and returns `oldChild`.
- **Node replaceChild(Node newChild, Node oldChild)** replaces `oldChild` with `newChild` and returns `oldChild`.
- **Node appendChild(Node newChild)** adds `newChild` to the end of the current node's child nodes and returns `newChild`.

Finally, Node declares several utility methods, including **Node cloneNode(boolean deep)** (create and return a duplicate of the current node, recursively cloning its subtree when `true` is passed to `deep`), and **void normalize()** (descend the tree from the given node and merge all adjacent text nodes, deleting those text nodes that are empty).

**Tip**

To obtain an element node's attributes, first call Node's **NamedNodeMap getAttributes()** method. This method returns an `org.w3c.dom.NamedNodeMap` implementation when the node represents an element; otherwise, it returns null. Along with declaring methods for accessing these nodes by name (such as **Node getNamedItem(String name)**), NamedNodeMap declares **int getLength()** and **Node item(int index)** methods for returning all attribute nodes by index. You would then obtain the Node's name by calling a method such as **getNodeName()**.

Beyond inheriting Node's constants and methods, Document declares its own methods. For example, you can call Document's **String getXmlEncoding()**, **boolean getXmlStandalone()**, and **String getXmlVersion()** methods to return the XML declaration's encoding, standalone, and version attribute values, respectively.

Document declares three methods for locating one or more elements:

- **Element getElementById(String elementId)** returns the element that has an `id` attribute (as in `<img id=...>`) matching the value specified by `elementId`.
- **NodeList getElementsByTagName(String tagname)** returns a nodelist of a document's elements (in document order) matching the specified `tagName`.
- **NodeList getElementsByTagNameNS(String namespaceURI, String localName)** is essentially the same as the second method except that only elements matching the given `localName` and `namespaceURI` are returned in the nodelist. Pass "*" to `namespaceURI` to match all namespaces; pass "*" to `localName` to match all local names.
The returned element node and each element node in the list implement the org.w3c.dom.Element interface. This interface declares methods to return node lists of descendent elements in the tree, attributes associated with the element, and more. For example, String getAttribute(String name) returns the value of the attribute identified by name, whereas Attr getAttributeNode(String name) returns an attribute node by name. The returned node is an implementation of the org.w3c.dom.Attr interface.

You now have enough information to explore an application for parsing an XML document and outputting the element and attribute information from the resulting DOM tree. Listing 15-14 presents this application's source code.

Listing 15-14. DOMDemo (Version 1)

```java
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class DOMDemo
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java DOMDemo xmlfile");
            return;
        }
        try
        {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            dbf.setNamespaceAware(true);
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse(args[0]);
            System.out.printf("Version = %s%n", doc.getXmlVersion());
            System.out.printf("Encoding = %s%n", doc.getXmlEncoding());
            System.out.printf("Standalone = %b%n", doc.getXmlStandalone());
            if (doc.hasChildNodes())
            {
                NodeList nl = doc.getChildNodes();
                for (int i = 0; i < nl.getLength(); i++)
                {
                    Node n = nl.item(i);
                    if (n.getNodeType() == Node.ELEMENT_NODE)
                    {
                        Element e = (Element) n;
                        System.out.printf("Element %s, Attribute %s%n", e.getNodeName(), e.getAttribu...
Node node = nl.item(i);
    if (node.getNodeType() == Node.ELEMENT_NODE)
        dump((Element) node);

} }

DOMDemo's main() method first verifies that one command line argument (the name of an XML
document) has been specified. It then creates a document builder factory, informs the factory that it
wants a namespace-aware document builder, and has the factory return this document builder.
Continuing, `main()` parses the document into a node tree; outputs the XML declaration's version number, encoding, and standalone attribute values; and recursively dumps all element nodes (starting with the root node) and their attribute values.

Notice the use of `getNodeType()` in one part of this listing and `instanceof` in another part. The `getNodeType()` method call isn't necessary (it's only present for demonstration) because `instanceof` can be used instead. However, the cast from `Node` type to `Element` type in the `dump()` method calls is necessary.

Compile this source code (`javac DOMDemo.java`), and run the application to dump Listing 15-3's article XML content as follows:

```
java DOMDemo article.xml
```

You should observe the following output:

```
Version = 1.0
Encoding = null
Standalone = false

Element: article, article, null, null
   Attribute lang = en
   Attribute title = The Rebirth of JavaFX
Element: abstract, abstract, null, null
Element: code-inline, code-inline, null, null
Element: body, body, null, null

Each Element-prefixed line outputs the node name, followed by the local name, followed by the namespace prefix, followed by the namespace URI. The node and local names are identical because namespaces aren’t being used. For the same reason, the namespace prefix and namespace URI are null.

Continuing on, execute the following command line to dump Listing 15-5's recipe content:

```
java DOMDemo recipe.xml
```

This time, you observe the following output, which includes namespace information:

```
Version = 1.0
Encoding = null
Standalone = false

Element: h:html, html, h, http://www.w3.org/1999/xhtml
   Attribute xmlns:h = http://www.w3.org/1999/xhtml
   Attribute xmlns:r = http://www.tutortutor.ca/
Element: h:head, head, h, http://www.w3.org/1999/xhtml
Element: h:body, body, h, http://www.w3.org/1999/xhtml
```
Creating XML Documents

DocumentBuilder declares the abstract Document newDocument() method for creating a document tree. The returned Document object declares various “create” and other methods for creating this tree. For example, Element createElement(String tagName) creates an element named by tagName, returning a new Element object with the specified name but with its local name, prefix, and namespace URI set to null.

Listing 15-15 presents another version of the DOMDemo application that briefly demonstrates the creation of a document tree.

Listing 15-15. DOMDemo (Version 2)

```java
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;

public class DOMDemo
{
    public static void main(String[] args)
    {
        try
        {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.newDocument();
            // Create the root element.
            Element root = doc.createElement("movie");
            doc.appendChild(root);
            // Create name child element and add it to the root.
            Element name = doc.createElement("name");
            root.appendChild(name);
        } catch (ParserConfigurationException e) {
            System.out.println("Error creating document: "+ e.getMessage());
        } catch (FactoryConfigurationError e) {
            System.out.println("Error creating factory: "+ e.getMessage());
        } catch (Exception e) {
            System.out.println("Error creating document: "+ e.getMessage());
        }
    }
}
```
DOMDemo creates Listing 15-2’s movie document. It uses Document’s createElement() method to create the root movie element and movie’s name and language child elements. It also uses Document’s Text createTextNode(String data) method to create text nodes that are attached to the name and language nodes. Notice the calls to Node’s appendChild() method to append child nodes (such as name) to parent nodes (such as movie).
After creating this tree, DOMDemo outputs the tree’s element nodes and other information. This output appears as follows:

Version = 1.0
Encoding = null
Standalone = false

Element: movie, null, null, null
Element: name, null, null, null
Text: Le Fabuleux Destin d’Amélie Poulain
Element: language, null, null, null
Text: français

The output is pretty much as expected, but there’s one problem: the XML declaration’s encoding attribute hasn’t been set to ISO-8859-1. You cannot accomplish this task via the DOM API. Instead, you need to use the XSLT API. While exploring XSLT, you’ll learn how to set the encoding attribute, and you’ll also learn how to output this tree to an XML document file.

However, there’s one more document-parsing API to explore (and a tour of the XPath API to take) before we turn our attention to XSLT.

Parsing XML Documents with XMLPULL V1

SAX is an example of a push parser, which pushes parsing events to an application. The application provides a handler that responds to these events. The parser invokes the handler’s callback methods to execute application-specific code as XML constructs are detected. Although push parsing is simple to implement, it often results in applications that are hard to write and debug.

To avoid SAX’s nonintuitiveness, you can use DOM to parse a document into an in-memory tree of nodes. However, the maximum size of this tree (and document) is constrained by available memory.

There’s a third option for parsing XML documents that overcomes SAX’s and DOM’s disadvantages. A pull parser lets an application pull parsed XML constructs, one at a time, from the parser when these constructs are needed. Unlike a push parser, which drives the application, a pull parser is driven by the application. Applications that use pull parsing are easier to write and debug.

Java 6 formally introduced pull parsing via Streaming API for XML (StAX). StAX supports two forms of pull parsing: event-based and stream-based. It also lets you create XML documents. Google ignored StAX and chose the simpler XMLPULL V1 for its pull parser API. Unlike StAX, XMLPULL V1 supports event-based pull parsing only. Also, it doesn’t let you create XML documents.

Because XMLPULL V1 isn’t included with Oracle Java, you’ll need to download a JAR file before trying out this section’s example application. Complete the following steps to accomplish this task.

1. Point your browser to www.java2s.com/Code/Jar/x/
   Downloadxmlpullxpp3114cjar.htm.

2. Click the xmlpull/xmlpull-xpp3-1.1.4c.jar.zip (109 k) (or equivalent) link.

You should end up with an xmlpull-xpp3-1.1.4c.jar file (or a later version of this file), which you will add to your CLASSPATH environment variable (or specify via the -classpath/-cp command-line option) when compiling or running this section’s example application.
CHAPTER 15: Parsing, Creating, and Transforming XML Documents

XMLPULL V1 consists of an `org.xmlpull.v1` package that contains the following types:

- **XmlPullParserFactory**: A class that lets you create and return instances of the XMLPULL parser.
- **XmlPullParser**: An interface that defines parsing functionality provided in XMLPULL V1.
- **XmlPullParserException**: A class that signals XML parsing problems. This exception is widely thrown throughout the API.

Listing 15-16 presents the source code for a simple application that demonstrates these types.

Listing 15-16. **XMLPPDemo**

```
import java.io.FileReader;
import java.io.IOException;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;
import org.xmlpull.v1.XmlPullParserFactory;

public class XMLPPDemo
{
    public static void main (String args[])
        throws IOException, XmlPullParserException
    {
        if (args.length != 1)
        {
            System.err.println("usage: java XMLPPDemo xmlfilespec");
            return;
        }

        XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
        factory.setNamespaceAware(true);
        XmlPullParser xpp = factory.newPullParser();
        xpp.setInput(new FileReader(args[0]));
        int eventType = xpp.getEventType();
        while (eventType != XmlPullParser.END_DOCUMENT)
        {
            switch (eventType)
            {
            case XmlPullParser.START_DOCUMENT:
                System.out.println("Start document");
                break;

            case XmlPullParser.START_TAG:
                System.out.println("Start tag " + xpp.getName());
                break;
            }
        }
    }
}
```
case XmlPullParser.TEXT:
    System.out.println("Text " + xpp.getText());
    break;

case XmlPullParser.END_TAG:
    System.out.println("End tag " + xpp.getName());
}
eventType = xpp.next();
}
}

Listing 15-16 describes an XMLPPDemo application that parses the XML file identified by its single command-line argument.

After verifying that a single command-line argument has been specified, main() invokes XmlPullParserFactory's newInstance() class method to create and return a new XmlPullParserFactory instance for creating XML pull parsers. XmlPullParserException is thrown when an XmlPullParserFactory implementation class cannot be found.

Next, void setNamespaceAware(boolean awareness) is invoked with a true argument to tell the XML pull parser factory to create only a parser that supports XML namespaces.

At this point, XmlPullParser newPullParser() is invoked to create and return a new XmlPullParser instance using the currently-configured factory parameters.

XmlPullParser declares a void setInput(Reader reader) method that sets the parser's input source to the specified reader. In this example, the command-line argument is passed to a java.io.FileReader constructor and the FileReader instance is passed to setInput(), to signify that XML content will be read from the file.

XmlPullParser declares a int getEventType() method that returns the current event type. The following integer-based event type constants are provided:

- START_DOCUMENT: The parser is at the very beginning of the document and nothing has yet been read. This is the initial event type after setInput() is called.
- START_TAG: A tag has been read. Invoke XmlPullParser's String getName() method to return the tag's local name and String getPrefix() to return any prefix. You can invoke various "getAttribute" methods to return attribute information.
- TEXT: Character data has been read and can be obtained by calling XmlPullParser's String getText() method.
- END_TAG: The end tag has just been read.
- END_DOCUMENT: The parser has reached the logical end of the document; attempting to read another item results in XmlPullParserException being thrown.

main()'s final task is to enter a while loop that executes while the current event type doesn't equal END_DOCUMENT. The loop first outputs information about the event. It then invokes XMLPullParser's int next() method to obtain the next parser event.
Execute the following command line to compile Listing 15-16:

```bash
javac -cp xmlpull-xpp3-1.1.4c.jar XMLPPDemo.java
```

Assuming that Listing 15-3’s `article.xml` file is located in the current directory, execute the following command line to parse this file:

```bash
java -cp xmlpull-xpp3-1.1.4c.jar:. XMLPPDemo article.xml
```

You should observe the following output:

```
Start document
Start tag article
Text

Start tag abstract
Text
    JavaFX 2.0 marks a significant milestone in the history of JavaFX. Now that
    Sun Microsystems has passed the torch to Oracle, we have seen the demise of
    JavaFX Script and the emerge of Java APIs (such as

Start tag code-inline
Text javafx.application.Application
End tag code-inline
Text ) for interacting
    with this technology. This article introduces you to this new flavor of
    JavaFX, where you learn about JavaFX 2.0 architecture and key APIs.

End tag abstract
Text

Start tag body
Text

End tag body
Text

End tag article
```

Selecting XML Document Nodes with XPath

XPath is a non-XML declarative query language (defined by the W3C) for selecting an XML document’s infoset items as one or more nodes. For example, you can use XPath to locate Listing 15-1’s third ingredient element and return this element node.

XPath is often used to simplify access to a DOM tree’s nodes, and it is also used in the context of XSLT (discussed in the next section) where it’s typically employed to select those input document elements (via XPath expressions) that are to be copied to an output document. Java and Android support XPath 1.0, which is assigned package javax.xml.xpath.

In this section, I first acquaint you with the XPath language. I then demonstrate how XPath simplifies the selection of a DOM tree’s nodes. Lastly, I introduce three advanced XPath topics: namespace contexts, extension functions and function resolvers, and variables and variable resolvers.

XPath Language Primer

XPath views an XML document as a tree of nodes that starts from a root node. XPath recognizes seven kinds of nodes: element, attribute, text, namespace, processing instruction, comment, and document. XPath doesn’t recognize CDATA sections, entity references, or document type declarations.

Note A tree’s root node (a DOM Document instance) isn’t the same as a document’s root element. The root node contains the entire document, including the root element, any comments or processing instructions that appear before the root element’s start tag and any comments or processing instructions that appear after the root element’s end tag.

Location Path Expressions

XPath provides location path expressions for selecting nodes. A location path expression locates nodes via a sequence of steps starting from the context node (the root node or some other document node that’s the current node). The returned set of nodes, which is known as a nodeset, might be empty, or it might contain one or more nodes.

The simplest location path expression selects the document’s root node and consists of a single forward slash character (/). The next simplest location path expression is the name of an element, which selects all child elements of the context node that have that name. For example, ingredient refers to all ingredient child elements of the context node in Listing 15-1’s XML document. This XPath expression returns a set of three ingredient nodes when the context node is ingredients. However, if recipe or instructions happened to be the context node, ingredient wouldn’t return any nodes (ingredient is a child of ingredients only). When an expression starts with a forward slash (/), the expression represents an absolute path that starts from the root node. For example, expression /movie selects all movie child elements of the root node in Listing 15-2’s XML document.

Attributes are also handled by location path expressions. To select an element’s attribute, specify @ followed by the attribute’s name. For example, @qty selects the qty attribute node of the context node.
In most cases, you'll work with root nodes, element nodes, and attribute nodes. However, you might also need to work with namespace nodes, text nodes, processing-instruction nodes, and comment nodes. Unlike namespace nodes, which are typically handled by XSLT, you'll more likely need to process comments, text, and processing instructions. XPath provides comment(), text(), and processing-instruction() functions for selecting comment, text, and processing-instruction nodes.

The comment() and text() functions don't require arguments because comment and text nodes don't have names. Each comment is a separate comment node, and each text node specifies the longest run of text not interrupted by a tag. The processing-instruction() function may be called with an argument that identifies the target of the processing instruction. If called with no argument, all of the context node's processing-instruction child nodes are selected.

XPath provides three wildcards for selecting unknown nodes:

- * matches any element node regardless of the node's type. It doesn’t match attributes, text nodes, comments, or processing-instruction nodes. When you place a namespace prefix before the *, only elements belonging to that namespace are matched.
- node() is a function that matches all nodes.
- @* matches all attribute nodes.

**Note** XPath lets you perform multiple selections by using the vertical bar (|). For example, author/*|publisher/* selects the children of author and the children of publisher, and *|@* matches all elements and attributes, but doesn’t match text, comment, or processing-instruction nodes.

XPath lets you combine steps into compound paths by using the / character to separate them. For paths beginning with /, the first path step is relative to the root node; otherwise, the first path step is relative to another context node. For example, /movie/name starts with the root node, selects all movie element children of the root node, and selects all name children of the selected movie nodes. If you wanted to return all text nodes of the selected name elements, you would specify /movie/name/text().

Compound paths can include // to select nodes from all descendants of the context node (including the context node). When placed at the start of an expression, // selects nodes from the entire tree. For example, //ingredient selects all ingredient nodes in the tree.

As with filesystems that let you identify the current directory with a single period (.) and its parent directory with a double period (..), you can specify a single period to represent the current node and a double period to represent the parent of the current node. (You would typically use a single period in XSLT to indicate that you want to access the value of the currently-matched element.)

It might be necessary to narrow the selection of nodes returned by an XPath expression. For example, expression /recipe/ingredients/ingredient returns all ingredient nodes, but perhaps you only want to return the first ingredient node. You can narrow the selection by including predicates in the location path.

A *predicate* is a square bracket-delimited Boolean expression that's tested against each selected node. If the expression evaluates to true, that node is included in the set of nodes returned by the XPath expression; otherwise, the node isn’t included in the set. For example, /recipe/ingredients/ingredient[1] selects the first ingredient element that's a child of the ingredients element.
Predicates can include predefined functions (such as \texttt{last()} and \texttt{position()}), operators (such as -, <, and =), and other items. Consider the following examples:

- \texttt{/recipe/ingredients/ingredient[last()]} selects the last ingredient element that's a child of the ingredients element.
- \texttt{/recipe/ingredients/ingredient[last() - 1]} selects the next-to-last ingredient element that's a child of the ingredients element.
- \texttt{/recipe/ingredients/ingredient[position() < 3]} selects the first two ingredient elements that are children of the ingredients element.
- \texttt{//ingredient[@qty]} selects all ingredient elements (no matter where they're located) that have qty attributes.
- \texttt{//ingredient[@qty='1'] or //ingredient[@qty="1"]} selects all ingredient elements (no matter where they're located) that have qty attributes with value 1.

\textbf{Note} XPath predefines several functions for use with nodesets: \texttt{last()} returns a number identifying the last node, \texttt{position()} returns a number identifying a node's position, \texttt{count()} returns the number of nodes in its nodeset argument, \texttt{id()} selects elements by their unique IDs and returns a nodeset of these elements, \texttt{local-name()} returns the local part of the qualified name of the first node in its nodeset argument, \texttt{namespace-uri()} returns the namespace part of the qualified name of the first node in its nodeset argument, and \texttt{name()} returns the qualified name of the first node in its nodeset argument.

Although predicates are supposed to be Boolean expressions, the predicate might not evaluate to a Boolean value. For example, it could evaluate to a number or a string. XPath supports Boolean, number (IEEE 754 double precision floating-point values), and string expression types as well as a location path expression's nodeset type. If a predicate evaluates to a number, XPath converts that number to true when it equals the context node's position; otherwise, XPath converts that number to false. If a predicate evaluates to a string, XPath converts that string to true when the string isn't empty; otherwise, XPath converts that string to false. Finally, if a predicate evaluates to a nodeset, XPath converts that nodeset to true when the nodeset is nonempty; otherwise, XPath converts that nodeset to false.

\textbf{Note} The previously presented location path expression examples demonstrate XPath's abbreviated syntax. However, XPath also supports an unabbreviated syntax that's more descriptive of what's happening and is based on an axis specifier that indicates the navigation direction within the XML document's tree representation. For example, where \texttt{/movie/name} selects all movie child elements of the root node followed by all name child elements of the movie elements using the abbreviated syntax, \texttt{/child::movie/child::name} accomplishes the same task with the expanded syntax. Check out Wikipedia's “XPath” entry (\url{http://en.wikipedia.org/wiki/XPath_1.0}) for more information.
General Expressions

Location path expressions (which return nodesets) are one kind of XPath expression. XPath also supports general expressions that evaluate to Boolean (such as predicates), number, or string type; for example, \texttt{position()} = 2, 6.8, and "Hello". General expressions are often used in XSLT.

XPath Boolean values can be compared via relational operators \(<\), \(\leq\), \(\geq\), \(=\), and \(!=\). Boolean expressions can be combined by using operators \(\text{and}\) and \(\text{or}\). Also, XPath predefines the following functions:

- \texttt{boolean()} returns a Boolean value for a number, string, or nodeset.
- \texttt{not()} returns true when its Boolean argument is false and vice-versa.
- \texttt{true()} returns true.
- \texttt{false()} returns false.
- \texttt{lang()} returns true or false depending on whether the language of the context node (as specified by \texttt{xml:lang} attributes) is the same as or is a sublanguage of the language specified by the argument string.

XPath numeric values can be manipulated via operators \(+\), \(-\), \(*\), \texttt{div}, and \texttt{mod} (remainder)—forward slash cannot be used for division because it’s used to separate location steps. All five operators behave like their Java language counterparts. XPath also pre defines the following functions:

- \texttt{number()} converts its argument to a number.
- \texttt{sum()} returns the sum of the numeric values represented by the nodes in its nodeset argument.
- \texttt{floor()} returns the largest (closest to positive infinity) number that’s not greater than its number argument and that’s an integer.
- \texttt{ceiling()} returns the smallest (closest to negative infinity) number that’s not less than its number argument and that’s an integer.
- \texttt{round()} returns the number that’s closest to the argument and that’s an integer. When there are two such numbers, the one closest to positive infinity is returned.

XPath strings are ordered character sequences that are enclosed in single quotes or double quotes. A string literal cannot contain the same kind of quote that’s also used to delimit the string. For example, a string that contains a single quote cannot be delimited with single quotes. XPath provides the \(=\) and \(!=\) operators for comparing strings. XPath also predefines the following functions:

- \texttt{string()} converts its argument to a string.
- \texttt{concat()} returns a concatenation of its string arguments.
- \texttt{starts-with()} returns true when its first argument string starts with its second argument string (and otherwise returns false).
- \texttt{contains()} returns true when its first argument string contains its second argument string (and otherwise returns false).
substring-before() returns the substring of its first argument string that precedes the first occurrence of its second argument string in its first argument string or the empty string when its first argument string doesn’t contain its second argument string.

substring-after() returns the substring of its first argument string that follows the first occurrence of its second argument string in its first argument string or the empty string when its first argument string doesn’t contain its second argument string.

substring() returns the substring of its first (string) argument starting at the position specified in its second (number) argument with length specified in its third (number) argument.

string-length() returns the number of characters in its string argument (or the length of the context node when converted to a string in the absence of an argument).

normalize-space() returns the argument string with whitespace normalized by stripping leading and trailing whitespace and replacing sequences of whitespace characters by a single space (or performing the same action on the context node when converted to a string in the absence of an argument).

translate() returns its first argument string with occurrences of characters in its second argument string replaced by the character at the corresponding position in its third argument string.

**XPath and DOM**

Suppose you need someone in your home to purchase a bag of sugar. You could say, “Please buy me some sugar.” Alternatively, you could say the following: “Please open the front door. Walk down to the sidewalk. Turn left. Walk up the sidewalk for three blocks. Turn right. Walk up the sidewalk one block. Enter the store. Go to aisle 7. Walk two meters down the aisle. Pick up a bag of sugar. Walk to a checkout counter. Pay for the sugar. Retrace your steps home.” Most people would expect to receive the shorter instruction, right?

Traversing a DOM tree of nodes is similar to providing the longer sequence of instructions. In contrast, XPath lets you traverse this tree via a succinct instruction. To see this difference for yourself, consider a scenario where you have an XML-based contacts document that lists your various professional contacts. Listing 15-17 presents a trivial example of such a document.

**Listing 15-17. XML-Based Contacts Database**

```xml
<?xml version="1.0"?>
<contacts>
  <contact>
    <name>John Doe</name>
    <city>Chicago</city>
    <city>Denver</city>
  </contact>
</contacts>
```
CHAPTER 15: Parsing, Creating, and Transforming XML Documents

Listing 15-17 reveals a simple XML grammar consisting of a contacts root element that contains a sequence of contact elements. Each contact element contains one name element and one or more city elements (various contacts travel frequently and spend a lot of time in each city). (To keep the example simple, I'm not providing a DTD or a schema.)

Suppose you want to locate and output the names of all contacts that live at least part of each year in Chicago. Listing 15-18 presents the source code to a DOMSearch application that accomplishes this task with the DOM API.

Listing 15-18. Locating Chicago Contacts with the DOM API

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class DOMSearch {
    public static void main(String[] args)
    {
        try
        {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse("contacts.xml");
```
List<String> contactNames = new ArrayList<String>();
NodeList contacts = doc.getElementsByTagName("contact");
for (int i = 0; i < contacts.getLength(); i++)
{
    Element contact = (Element) contacts.item(i);
    NodeList cities = contact.getElementsByTagName("city");
    boolean chicago = false;
    for (int j = 0; j < cities.getLength(); j++)
    {
        Element city = (Element) cities.item(j);
        NodeList children = city.getChildNodes();
        StringBuilder sb = new StringBuilder();
        for (int k = 0; k < children.getLength(); k++)
        {
            Node child = children.item(k);
            if (child.getNodeType() == Node.TEXT_NODE)
                sb.append(child.getNodeValue());
        }
        if (sb.toString().equals("Chicago"))
        {
            chicago = true;
            break;
        }
    }
    if (chicago)
    {
        NodeList names = contact.getElementsByTagName("name");
        contactNames.add(names.item(0).getFirstChild().getNodeValue());
    }
}
for (String contactName: contactNames)
    System.out.println(contactName);
}
catch (IOException ioe)
{
    System.err.println("IOE: " + ioe);
}
catch (SAXException saxe)
{
    System.err.println("SAXE: " + saxe);
}
catch (FactoryConfigurationError fce)
{
    System.err.println("FCE: " + fce);
}
catch (ParserConfigurationException pce)
{
    System.err.println("PCE: " + pce);
}
}
After parsing contacts.xml and building the DOM tree, main() uses Document's getElementsByTagName() method to return a NodeList of contact element nodes. For each member of this list, main() extracts the contact element node, and uses this node with getElementsByTagName() to return a NodeList of the contact element node's city element nodes.

For each member of the cities list, main() extracts the city element node, and uses this node with getElementsByTagName() to return a NodeList of the city element node's child nodes; there's only a single child text node in this example, but the presence of a comment or processing instruction would increase the number of child nodes. For example, <city>Chicago<!--The windy city--></city> increases the number of child nodes to 2.

If the child's node type indicates that it's a text node, the child node's value (obtained via getNodeValue()) is stored in a string builder. Only one child node is stored in the string builder in this example. If the builder's contents indicate that Chicago has been found, the chicago flag is set to true and execution leaves the cities loop.

If the chicago flag is set when the cities loop exits, the current contact element node's getElementsByTagName() method is called to return a NodeList of the contact element node's name element nodes (of which there should only be one, and which I could enforce through a DTD or schema). It's now a simple matter to extract the first item from this list, call getFirstChild() on this item to return the text node (I assume that only text appears between <name> and </name>), and call getNodeValue() on the text node to obtain its value, which is then added to the contactNames list.

After compiling this source code, run the application. You should observe the following output:

John Doe
Bob Jones

Traversing the DOM's tree of nodes is a tedious exercise at best and is error-prone at worst. Fortunately, XPath can greatly simplify this situation.

Before writing the XPath equivalent of Listing 15-18, it helps to define a location path expression. For this example, that expression is //contact[city = "Chicago"]/name/text(), which uses a predicate to select all contact nodes that contain a Chicago city node, then select all child name nodes from these contact nodes, and finally select all child text nodes from these name nodes.

Listing 15-19 presents the source code to an XPathSearch application that uses this XPath expression and the XPath API to locate Chicago contacts.

Listing 15-19. Locating Chicago Contacts with the XPath API

```java
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathException;
import javax.xml.xpath.XPathFactory;

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathException;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;
```
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XPathSearch
{
    public static void main(String[] args)
    {
        try
        {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse("contacts.xml");
            XPathFactory xpf = XPathFactory.newInstance();
            XPath xp = xpf.newXPath();
            XPathExpression xpe;
            xpe = xp.compile("//contact[city = 'Chicago']/name/text()");
            Object result = xpe.evaluate(doc, XPathConstants.NODESET);
            NodeList nl = (NodeList) result;
            for (int i = 0; i < nl.getLength(); i++)
            {
                System.out.println(nl.item(i).getNodeValue());
            }
        }
        catch (IOException ioe)
        {
            System.err.println("IOE: " + ioe);
        }
        catch (SAXException saxe)
        {
            System.err.println("SAXE: " + saxe);
        }
        catch (FactoryConfigurationError fce)
        {
            System.err.println("FCE: " + fce);
        }
        catch (ParserConfigurationException pce)
        {
            System.err.println("PCE: " + pce);
        }
        catch (XPathException xpe)
        {
            System.err.println("XPE: " + xpe);
        }
    }
}

After parsing contacts.xml and building the DOM tree, main() instantiates XPathFactory by calling its XPathFactory.newInstance() method. The resulting XPathFactory instance can be used to set features (such as secure processing, to process XML documents securely) by calling its void setFeature(String name, boolean value) method, to create an XPath object by calling its XPath newXPath() method, and more.
XPath declares an XPathExpression compile(String expression) method for compiling the specified expression (an XPath expression) and returning the compiled expression as an instance of a class that implements the XPathExpression interface. This method throws XPathExpressionException (a subclass of XPathException) when the expression cannot be compiled.

XPath also declares several overloaded evaluate() methods for immediately evaluating an expression and returning the result. Because it can take time to evaluate an expression, you might choose to compile a complex expression first (to boost performance) when you plan to evaluate this expression many times.

After compiling the expression, main() calls XPathExpression's Object evaluate(Object item, QName returnType) method to evaluate the expression. The first argument is the context node for the expression, which happens to be a Document instance in the example. The second argument specifies the kind of object returned by evaluate() and is set to XPathConstants.NODESET, a qualified name for the XPath 1.0 nodeset type, which is implemented via DOM's NodeList interface.

**Note** The XPath API maps XPath's Boolean, number, string, and nodeset types to Java's java.lang.Boolean, java.lang.Double, String, and NodeList types, respectively. When calling an evaluate() method, you specify XPath types via XPathConstants constants (BOOLEAN, NUMBER, STRING, and NODESET), and the method takes care of returning an object of the appropriate type. XPathConstants also declares a NODE constant, which doesn’t map to a Java type. Instead, it's used to tell evaluate() that you only want the resulting nodeset to contain a single node.

After casting Object to NodeList, main() uses this interface's getLength() and item() methods to traverse the nodelist. For each item in this list, getNodeValue() is called to return the node's value, which is subsequently output. XPathSearch generates the same output as DOMSearch.

**Advanced XPath**

The XPath API provides three advanced features to overcome limitations with the XPath 1.0 language. These features are namespace contexts, extension functions and function resolvers, and variables and variable resolvers.

**Namespace Contexts**

When an XML document's elements belong to a namespace (including the default namespace), any XPath expression that queries the document must account for this namespace. For nondefault namespaces, the expression doesn’t need to use the same namespace prefix; it only needs to use the same URI. However, when a document specifies the default namespace, the expression must use a prefix even though the document doesn’t use a prefix.

To appreciate this situation, suppose Listing 15-17's <contacts> tag was declared as follows to introduce a default namespace: <contacts xmlns="http://www.tutortutor.ca/">. Furthermore, suppose that Listing 15-19 included dbf.setNamespaceAware(true); after the line that instantiates DocumentBuilderFactory. If you were to run the revised XPathSearch application against the revised contacts.xml file, you wouldn’t see any output.
You can correct this problem by implementing javax.xml.namespace.NamespaceContext to map an arbitrary prefix to the namespace URI, and then registering this namespace context with the XPath instance. Listing 15-20 presents a minimal implementation of the NamespaceContext interface.

Listing 15-20. Minimally Implementing NamespaceContext

```java
import java.util.Iterator;
import javax.xml.XMLConstants;
import javax.xml.namespace.NamespaceContext;

public class NSContext implements NamespaceContext {
    @Override
    public String getNamespaceURI(String prefix) {
        if (prefix == null)
            throw new IllegalArgumentException("prefix is null");
        else
            if (prefix.equals("tt"))
                return "http://www.tutor.ca/";
            else
                return null;
    }

    @Override
    public String getPrefix(String uri) {
        return null;
    }

    @Override
    public Iterator getPrefixes(String uri) {
        return null;
    }
}
```

The getNamespaceURI() method is passed a prefix argument that must be mapped to a URI. If this argument is null, an IllegalArgumentException object must be thrown (according to the Java documentation). When the argument is the desired prefix value, the namespace URI is returned.

After instantiating the XPath class, you would instantiate NSContext and register this instance with the XPath instance by calling XPath's void setNamespaceContext(NamespaceContext nsContext) method. For example, you would specify `xp.setNamespaceContext(new NSContext());` after `XPath xp = xpf.newXPath();` to register the NSContext instance with xp.
All that's left to accomplish is to apply the prefix to the XPath expression, which now becomes
//tt:contact[tt:city='Chicago']/tt:name/text() because the contact, city, and name elements
are now part of the default namespace, whose URI is mapped to arbitrary prefix tt in the NSContext
instance's getNamespaceURI() method.

Compile and run the revised XPathSearch application, and you'll see John Doe followed by Bob Jones
on separate lines.

**Extension Functions and Function Resolvers**

The XPath API lets you define functions (via Java methods) that extend XPath's predefined function
repertoire by offering new features not already provided. These Java methods cannot have side
effects because XPath functions can be evaluated multiple times and in any order. Furthermore, they
cannot override predefined functions; a Java method with the same name as a predefined function is
never executed.

Suppose you modify Listing 15-17's XML document to include a birth element that records a contact’s
date of birth information in YYYY-MM-DD format. Listing 15-21 shows the resulting XML file.

**Listing 15-21. XML-Based Contacts Database with Birth Information**

```xml
<?xml version="1.0"?>
<contacts xmlns="http://www.tutortutor.ca/">
  <contact>
    <name>John Doe</name>
    <birth>1953-01-02</birth>
    <city>Chicago</city>
    <city>Denver</city>
  </contact>
  <contact>
    <name>Jane Doe</name>
    <birth>1965-07-12</birth>
    <city>New York</city>
  </contact>
  <contact>
    <name>Sandra Smith</name>
    <birth>1976-11-22</birth>
    <city>Denver</city>
    <city>Miami</city>
  </contact>
  <contact>
    <name>Bob Jones</name>
    <birth>1958-03-14</birth>
    <city>Chicago</city>
  </contact>
</contacts>
```

Now suppose that you want to select contacts based on birth information. For example, you only
want to select contacts whose date of birth is greater than 1960-01-01. Because XPath doesn't
provide this function for you, you decide to declare a date() extension function. Your first step is to
declare a Date class that implements the XPathFunction interface (see Listing 15-22).
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.List;
import javax.xml.xpath.XPathFunction;
import javax.xml.xpath.XPathFunctionException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class Date implements XPathFunction
{
  private final static ParsePosition POS = new ParsePosition(0);
  private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-mm-dd");

  @Override
  public Object evaluate(List args) throws XPathFunctionException
  {
    if (args.size() != 1)
      throw new XPathFunctionException("Invalid number of arguments");
    String value;
    Object o = args.get(0);
    if (o instanceof NodeList)
    {
      NodeList list = (NodeList) o;
      value = list.item(0).getTextContent();
    }
    else
    if (o instanceof String)
      value = (String) o;
    else
      throw new XPathFunctionException("Cannot convert argument type");
    POS.setIndex(0);
    return sdf.parse(value, POS).getTime();
  }
}

XPathFunction declares a single Object evaluate(List args) method that XPath calls when it needs to execute the extension function. evaluate() is passed a java.util.List of objects that describe the arguments that were passed to the extension function by the XPath evaluator. Furthermore, this method returns a value of a type appropriate to the extension function (date()’s long integer return type is compatible with XPath’s number type).

The date() extension function is intended to be called with a single argument, which is either of type nodeset or of type string. This extension function throws XPathFunctionException when the number of arguments (as indicated by the list’s size) isn’t equal to 1.
When the argument is of type NodeList (a nodeset), the textual content of the first node in the nodeset is obtained; this content is assumed to be a date value in YYYY-MM-DD format (for brevity, I’m overlooking error checking). When the argument is of type String, it’s assumed to be a date value in this format. Any other type of argument results in a thrown XPathFunctionException instance.

Date comparison is simplified by converting the date to a milliseconds value. This task is accomplished with the help of the java.text.SimpleDateFormat and java.text.ParsePosition classes. After resetting the ParsePosition object’s index (via setIndex(0)), SimpleDateFormat’s Date parse(String text, ParsePosition pos) method is called to parse the string according to the pattern established when SimpleDateFormat was instantiated, and starting from the parse position identified by the ParsePosition index. This index is reset before the parse() method call because parse() updates this object’s index.

The parse() method returns a java.util.Date instance whose long getTime() method is called to return the number of milliseconds represented by the parsed date. (I discuss SimpleDateFormat, ParsePosition, and Date in Chapter 16.)

After implementing the extension function, you need to create a function resolver, which is an object whose class implements the XPathFunctionResolver interface, and which tells the XPath evaluator about the extension function (or functions). Listing 15-23 presents the DateResolver class.

Listing 15-23. A Function Resolver for the date() Extension Function

```java
import javax.xml.namespace.QName;
import javax.xml.xpath.XPathFunction;
import javax.xml.xpath.XPathFunctionResolver;

public class DateResolver implements XPathFunctionResolver
{
    private static final QName name = new QName("http://www.tutortutor.ca/",
                                                    "date", "tt");

    @Override
    public XPathFunction resolveFunction(QName name, int arity)
    {
        if (name.equals(this.name) && arity == 1)
            return new Date();
        return null;
    }
}
```

XPathFunctionResolver declares a single XPathFunction resolveFunction(QName functionName, int arity) method that XPath calls to identify the name of the extension function and obtain an instance of a Java object whose evaluate() method implements the function.

The functionName parameter identifies the function’s qualified name because all extension functions must live in a namespace and must be referenced via a prefix (which doesn’t have to match the prefix in the document). As a result, you must also bind a namespace to the prefix via a namespace context (as demonstrated previously). The arity parameter identifies the number of arguments that the extension function accepts and is useful when overloading extension functions. If the functionName and arity values are acceptable, the extension function’s Java class is instantiated and returned; otherwise, null is returned.
Finally, the function resolver class is instantiated and registered with the XPath instance by calling XPath's void setXPathFunctionResolver(XPathFunctionResolver resolver) method.

The following excerpt from Version 3 of this chapter's XPathSearch application (in this book's code archive) demonstrates all of these tasks in order to use date() in XPath expression //tt:contact[tt:date(tt:birth) > tt:date('1960-01-01')]/tt:name/text(), which returns only those contacts whose date of birth is greater than 1960-01-01 (Jane Doe followed by Sandra Smith):

```java
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("contacts.xml");
XPathFactory xpf = XPathFactory.newInstance();
XPath xp = xpf.newXPath();
xp.setNamespaceContext(new NSContext());
xp.setXPathFunctionResolver(new DateResolver());
XPathExpression xpe;
String expr;
expr = "//tt:contact[tt:date(tt:birth) > tt:date('1960-01-01')]/tt:name/text()";
xpe = xp.compile(expr);
Object result = xpe.evaluate(doc, XPathConstants.NODESET);
NodeList nl = (NodeList) result;
for (int i = 0; i < nl.getLength(); i++)
   System.out.println(nl.item(i).getNodeValue());
```

**Variables and Variable Resolvers**

All of the previously-specified XPath expressions have been based on literal text. XPath also lets you specify variables to parameterize these expressions in a similar manner to using variables with SQL prepared statements.

A variable appears in an expression by prefixing its name (which may or may not have a namespace prefix) with a $. For example, /a/b[@c = $d]/text() is an XPath expression that selects all a elements of the root node, and all of a's b elements that have c attributes containing the value identified by variable $d, and returns the text of these b elements. This expression corresponds to Listing 15-24's XML document.


```xml
<?xml version="1.0"?>
<a>
   <b c="x">b1</b>
   <b>b2</b>
   <b c="y">b3</b>
   <b>b4</b>
   <b c="x">b5</b>
</a>
```
To specify variables whose values are obtained during expression evaluation, you must register a variable resolver with your XPath object. A variable resolver is an instance of a class that implements the XPathVariableResolver interface in terms of its Object resolveVariable(QName variableName) method, and which tells the evaluator about the variable (or variables).

The variableName parameter contains the qualified name of a variable's name. (Remember that a variable name may be prefixed with a namespace prefix.) This method verifies that the qualified name appropriately names the variable and then returns its value.

After creating the variable resolver, you register it with the XPath instance by calling XPath's void setXPathVariableResolver(XPathVariableResolver resolver) method.

The following excerpt from Version 4 of this chapter's XPathSearch application (in this book's code archive) demonstrates all of these tasks in order to specify $d$ in XPath expression /a/b[@c=$d]/text(), which returns b1 followed by b5. It assumes that Listing 15-24 is stored in a file named example.xml.

```java
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("example.xml");
XPathFactory xpf = XPathFactory.newInstance();
XPath xp = xpf.newXPath();
XPathVariableResolver xpvr;
xpvr = new XPathVariableResolver()
    {
        @Override
        public Object resolveVariable(QName varname)
        {
            if (varname.getLocalPart().equals("d"))
                return "x";
            else
                return null;
        }
    };
xp.setXPathVariableResolver(xpvr);
XPathExpression xpe;
xpe = xp.compile("/a/b[@c = $d]/text()");
Object result = xpe.evaluate(doc, XPathConstants.NODESET);
 NodeList nl = (NodeList) result;
for (int i = 0; i < nl.getLength(); i++)
    System.out.println(nl.item(i).getNodeValue());
```

Caution When you qualify a variable name with a namespace prefix (as in $ns:d), you must also register a namespace context to resolve the prefix.
Transforming XML Documents with XSLT

Extensible Stylesheet Language (XSL) is a family of languages for transforming and formatting XML documents. XSL Transformation (XSLT) is the XSL language for transforming XML documents to other formats, such as HTML (for presenting an XML document’s content via a web browser).

XSLT accomplishes its work by using XSLT processors and stylesheets. An XSLT processor is a software component that applies an XSLT stylesheet (an XML-based template consisting of content and transformation instructions) to an input document (without modifying the document), and copies the transformed result to a result tree, which can be output to a file or output stream, or even piped into another XSLT processor for additional transformations. Figure 15-3 illustrates the transformation process.

![Figure 15-3. An XSLT processor transforms an XML input document into a result tree](image)

The beauty of XSLT is that you don’t need to develop custom software applications to perform the transformations. Instead, you simply create an XSLT stylesheet and input it along with the XML document needing to be transformed to an XSLT processor.

In this section, I first introduce you to Java’s XSLT API. I then present two demonstrations of XSLT’s usefulness.

Exploring the XSLT API

Java implements XSLT through the types in the javax.xml.transform, javax.xml.transform.dom, javax.xml.transform.sax, and javax.xml.transform.stream packages. Oracle also provides a javax.xml.transform.stax package, which isn’t supported by Android.

The javax.xml.transform package defines the generic APIs for processing transformation instructions and for performing a transformation from a source (where the XSLT processor's input originates) to a result (where the processor's output is sent). The remaining packages define the APIs for obtaining different kinds of sources and results.

The javax.xml.transform.TransformerFactory class is the starting point for working with XSLT. You instantiate TransformerFactory by calling one its newInstance() methods. The following example uses TransformerFactory's TransformerFactory newInstance() class method to create the factory:

```java
TransformerFactory tf = TransformerFactory.newInstance();
```
Behind the scenes, newInstance() follows an ordered lookup procedure to identify the TransformerFactory implementation class to load. This procedure first examines the javax.xml.transform.TransformerFactory system property, and lastly chooses the Java platform's default TransformerFactory implementation class when no other class is found. If an implementation class isn't available (perhaps the class identified by the javax.xml.transform.TransformerFactory system property doesn't exist) or cannot be instantiated, newInstance() throws an instance of the javax.xml.transform.TransformerFactoryConfigurationError class. Otherwise, it instantiates the class and returns its instance.

After obtaining a TransformerFactory instance, you can call various configuration methods to configure the factory. For example, you could call TransformerFactory's void setFeature(String name, boolean value) method to enable a feature (such as secure processing, to transform XML documents securely).

Following the factory's configuration, call one of its newTransformer() methods to create and return instances of the javax.xml.transform.Transformer class. The following example calls Transformer newTransformer() to accomplish this task:

```java
Transformer t = tf.newTransformer();
```

The no-argument newTransformer() method copies source input to the destination without making any changes. This kind of transformation is known as the identity transformation.

To change input, you need to specify a stylesheet, and you accomplish this task by calling the factory's Transformer newTransformer(Source source) method, where the javax.xml.transform.Source interface describes a source for the stylesheet. The following example demonstrates this task:

```java
Transformer t = tf.newTransformer(new StreamSource(new FileReader("recipe.xsl")));
```

This example creates a transformer that obtains a stylesheet from a file named recipe.xsl via a javax.xml.transform.stream.StreamSource instance connected to a file reader. It's customary to use the .xsl or .xslt extension to identify XSLT stylesheet files.

The newTransformer() methods throw javax.xml.transform.TransformerConfigurationException when they cannot return a Transformer instance that corresponds to the factory configuration.

After obtaining a Transformer instance, you can call its void setOutputProperty(String name, String value) method to influence a transformation. The javax.xml.transform.OutputKeys class declares constants for frequently used keys. For example, OutputKeys.METHOD is the key for specifying the method for outputting the result tree (as XML, HTML, plain text, or something else).

**Tip** To set multiple properties in a single method call, create a java.util.Properties object and pass this object as an argument to Transformer's void setOutputProperties(Properties prop) method. Properties set by setOutputProperty() and setOutputProperties() override the stylesheet's xsl:output instruction settings.
Before you can perform a transformation, you need to obtain instances of classes that implement the Source and javax.xml.transform.Result interfaces. You then pass these instances to Transformer's void transform(Source xmlSource, Result outputTarget) method, which throws an instance of the javax.xml.transform.TransformerException class when a problem arises during the transformation.

The following example shows you how to obtain a source and a result, and perform the transformation:

```java
Source source = new DOMSource(doc);
Result result = new StreamResult(System.out);
t.transform(source, result);
```

The first line instantiates the javax.xml.transform.dom.DOMSource class, which acts as a holder for a DOM tree rooted in the Document object specified by doc. The second line instantiates the javax.xml.transform.stream.StreamResult class, which acts as a holder for the standard output stream, to which the transformed data items are sent. The third line reads data from the Source instance and outputs transformed data to the Result instance.

**Tip** Although Java's default transformers support the various Source and Result implementation classes located in the javax.xml.transform.dom, javax.xml.transform.sax, javax.xml.transform.stax (not in Android), and javax.xml.transform.stream packages, a nondefault transformer (perhaps specified via the javax.xml.transform.TransformerFactory system property) might be more limited. For this reason, each Source and Result implementation class declares a FEATURE string constant that can be passed to TransformerFactory's boolean getFeature(String name) method. This method returns true when the Source or Result implementation class is supported. For example, tf.getFeature(StreamSource.FEATURE) returns true when stream sources are supported. The javax.xml.transform.sax.SAXTransformerFactory class provides additional SAX-specific factory methods that you can use, but only when the TransformerFactory instance is also an instance of this class. To help you make the determination, SAXTransformerFactory also declares a FEATURE string constant that you can pass to getFeature(). For example, tf.getFeature(SAXTransformerFactory.FEATURE) returns true when the transformer factory referenced from tf is an instance of SAXTransformerFactory.

Most XML API interface instances and the factories that return them are not thread-safe. This situation also applies to transformers. Although you can reuse the same transformer multiple times on the same thread, you cannot access the transformer from multiple threads. This problem can be solved for transformers by using instances of classes that implement the javax.xml.transform.Templates interface. The Java documentation for this interface has this to say: **Templates must be threadsafe for a given instance over multiple threads running concurrently, and they may be used multiple times in a given session.** Along with promoting thread safety, Templates instances can improve performance because they represent compiled XSLT stylesheets.
The following example shows how you might perform a transformation without a Templates object:

```java
TransformerFactory tf = TransformerFactory.newInstance();
StreamSource ssStyleSheet = new StreamSource(new FileReader("recipe.xsl"));
Transformer t = tf.newTransformer(ssStyleSheet);
t.transform(new DOMSource(doc), new StreamResult(System.out));
```

You cannot access `t`'s transformer from multiple threads. In contrast, the following example shows you how to construct a transformer from a Templates object so that it can be accessed from multiple threads:

```java
TransformerFactory tf = TransformerFactory.newInstance();
StreamSource ssStyleSheet = new StreamSource(new FileReader("recipe.xsl"));
Templates te = tf.newTemplates(ssStylesheet);
Transformer t = te.newTransformer();
t.transform(new DOMSource(doc), new StreamResult(System.out));
```

The differences are the call to `TransformerFactory`'s `Templates newTemplates(Source source)` method to create and return objects whose classes implement the `Templates` interface and the call to this interface's `Transformer newTransformer()` method to obtain the `Transformer` instance.

### Demonstrating the XSLT API

Listing 15-15 presents a `DOMDemo` application that creates a DOM document tree based on Listing 15-2's movie XML document. Unfortunately, you cannot use the DOM API to assign ISO-8859-1 to the XML declaration's encoding attribute. Also, you cannot use DOM to output this tree to a file or other destination. However, you can overcome these problems with XSLT, as demonstrated in the following excerpt from Version 1 of this chapter's `XSLTDemo` application (in this book's code archive):

```java
TransformerFactory tf = TransformerFactory.newInstance();
Transformer t = tf.newTransformer();
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "3");
Source source = new DOMSource(doc);
Result result = new StreamResult(System.out);
t.transform(source, result);
```

After creating a transformer factory and obtaining a transformer from this factory, four output properties are specified to influence the transformation. `OutputKeys.METHOD` specifies that the result tree will be written out as XML, `OutputKeys.ENCODING` specifies that ISO-8859-1 will be the value of the XML declaration's encoding attribute, and `OutputKeys.INDENT` specifies that the transformer can output additional whitespace.

The additional whitespace is used to output the XML across multiple lines instead of on a single line. Because it would be nice to indicate the number of spaces for indenting lines of XML, and because this information cannot be specified via an `OutputKeys` property, the nonstandard "{http://xml.apache.org/xslt}indent-amount" property (property keys begin with brace-delimited URLs) is used to specify an appropriate value (such as 3 spaces). It's okay to specify this property in this example because Java's default XSLT implementation is based on Apache's XSLT implementation.
After setting properties, a source (the DOM document tree) and a result (the standard output stream) are obtained, and `transform()` is called to transform the source to the result.

Although this example shows you how to output a DOM tree and also how to specify an encoding value for the XML declaration of the resulting XML document, the example doesn’t really demonstrate the power of XSLT because (apart from setting the encoding attribute value) it performs an identity transformation. A more interesting example would take advantage of a stylesheet.

Consider a scenario where you want to convert Listing 15-1’s recipe document to an HTML document for presentation via a web browser. Listing 15-25 presents a stylesheet that a transformer can use to perform the conversion.

### Listing 15-25. An XSLT Stylesheet for Converting a Recipe Document to an HTML Document

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/recipe">
    <html>
        <head>
            <title>Recipes</title>
        </head>
        <body>
            <h2>
                <xsl:value-of select="normalize-space(title)"/>
            </h2>
            <h3>Ingredients</h3>
            <ul>
                <xsl:for-each select="ingredients/ingredient">
                    <li>
                        <xsl:value-of select="normalize-space(text())"/>
                        <xsl:if test="@qty"> (<xsl:value-of select="@qty"/>)</xsl:if>
                    </li>
                </xsl:for-each>
            </ul>
            <h3>Instructions</h3>
            <xsl:value-of select="normalize-space(instructions)"/>
        </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

Listing 15-25 reveals that a stylesheet is an XML document. Its root element is `stylesheet`, which identifies the standard namespace for stylesheets. It’s conventional to specify `xsl` as the namespace prefix for referring to XSLT instruction elements, although any prefix could be specified.
A stylesheet is based on template elements that control how an element and its content are converted. A template focuses on a single element that's identified via the `match` attribute. This attribute's value is an XPath location path expression, which matches all `recipe` child nodes of the root element node. Regarding Listing 15-1, only the single `recipe` root element will be matched and selected.

A template element can contain literal text and stylesheet instructions. For example, the `value-of` instruction in `<xsl:value-of select="normalize-space(title)"/>` specifies that the value of the `title` element (which is a child of the `recipe` context node) is to be retrieved and copied to the output. Because this text is surrounded by space and newline characters, XPath’s `normalize-string()` function is called to remove this whitespace before the title is copied.

XSLT is a powerful declarative language that includes control flow instructions such as `for-each` and `if`. In the context of `<xsl:for-each select="ingredients/ingredient">`, `for-each` causes all of the `ingredient` child nodes of the `ingredients` node to be selected and processed one at a time. For each node, `<xsl:value-of select="normalize-space(text())"/>` is executed to copy the content of the node, normalized to remove whitespace. Also, the `if` instruction in `<xsl:if test="@qty"> (<xsl:value-of select="@qty"/>)</xsl:if>` determines whether or not the `ingredient` node has a `qty` attribute, and (if so) copies a space character followed by this attribute’s value (surrounded by parentheses) to the output.

Note There’s a lot more to XSLT than can be demonstrated in this short example. To learn more about XSLT, I recommend that you check out Beginning XSLT 2.0 From Novice to Professional, by Jeni Tennison and published by Apress (www.apress.com/9781590593240). XSLT 2.0 is a superset of XSLT 1.0—Java 7 supports XSLT 1.0.

The following excerpt from Version 2 of this chapter’s XSLTDemo application (in this book’s code archive) shows how to write the Java code to process Listing 15-1 via Listing 15-24’s stylesheet:

```java
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("recipe.xml");
TransformerFactory tf = TransformerFactory.newInstance();
StreamSource ssStyleSheet;
ssStyleSheet = new StreamSource(new FileReader("recipe.xsl"));
Transformer t = tf.newTransformer(ssStyleSheet);
t.setOutputProperty(OutputKeys.METHOD, "html");
t.setOutputProperty(OutputKeys.INDENT, "yes");
Source source = new DOMSource(doc);
Result result = new StreamResult(System.out);
t.transform(source, result);
```

This excerpt reveals that the output method is set to `html`, and it also reveals that the resulting HTML should be indented. However, the output is only partly indented, as shown in Listing 15-26.
Listing 15-26. The HTML Equivalent of Listing 15-1’s Recipe Document

```html
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Recipes</title>
</head>
<body>
<h2>Grilled Cheese Sandwich</h2>
<h3>Ingredients</h3>
<ul>
<li>bread slice (2)</li>
<li>cheese slice</li>
<li>margarine pat (2)</li>
</ul>
<h3>Instructions</h3>
Place frying pan on element and select medium heat. For each bread slice, smear one pat of margarine on one side of bread slice. Place cheese slice between bread slices with margarine-smeared sides away from the cheese. Place sandwich in frying pan with one margarine-smeared side in contact with pan. Fry for a couple of minutes and flip. Fry other side for a minute and serve.
</body>
</html>
```

OutputKeys.INDENT and its "yes" value let you output the HTML across multiple lines as opposed to outputting the HTML on a single line. However, the XSLT processor performs no additional indentation, and it ignores attempts to specify the number of spaces to indent via code such as `t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "3");`.

### Note
An XSLT processor outputs a `<META>` tag when OutputKeys.METHOD is set to "html".

## EXERCISES

The following exercises are designed to test your understanding of Chapter 15’s content.

1. Define XML.
2. True or false: XML and HTML are descendants of SGML.
3. What is the XML declaration?
4. Identify the XML declaration’s three attributes. Which attribute is nonoptional?
5. True or false: An element always consists of a start tag followed by content followed by an end tag.
6. Following the XML declaration, an XML document is anchored in what kind of element?
7. What is mixed content?
8. What is a character reference? Identify the two kinds of character references.
9. What is a CDATA section? Why would you use it?
10. Define namespace.
11. What is a namespace prefix?
12. True or false: A tag’s attributes don’t need to be prefixed when those attributes belong to the element.
13. What is a comment? Where can a comment appear in an XML document?
14. Define processing instruction.
15. Identify the rules that an XML document must follow to be considered well formed.
16. What does it mean for an XML document to be valid?
17. A parser that performs validation compares an XML document to a grammar document. Identify the two commonly used grammar languages.
18. What is the general syntax for declaring an element in a DTD?
19. Which grammar language lets you create complex types from simple types?
20. Define SAX.
21. How do you obtain a SAX 2-based parser?
22. What is the purpose of the XMLReader interface?
23. How do you tell a SAX parser to perform validation?
24. Identify the four kinds of SAX-oriented exceptions that can be thrown when working with SAX.
25. What interface does a handler class implement to respond to content-oriented events?
26. Identify the three other core interfaces that a handler class is likely to implement.
27. Define ignorable whitespace.
28. True or false: void error(SAXParseException exception) is called for all kinds of errors.
29. What is the purpose of the org.xml.sax.helpers.DefaultHandler class?
30. What is an entity? What is an entity resolver?
31. Define DOM.
32. True or false: Java 7 and newer versions of Android support DOM Levels 1 and 2 only.
33. Identify the 12 different DOM nodes.
34. How do you obtain a document builder?
35. How do you use a document builder to parse an XML document?
36. True or false: Document and all other org.w3c.dom interfaces that describe different kinds of nodes are subinterfaces of the Node interface.
37. How do you use a document builder to create a new XML document?
38. When creating a new XML document, can you use the DOM API to specify the XML declaration’s encoding attribute?
39. What are a push parser and a pull parser?
40. True or false: Android uses Streaming API for XML (StAX) as its pull parser.
41. How do you obtain the pull parser?
42. How do you use the pull parser to parse an XML document?
43. Define XPath.
44. Where is XPath commonly used?
45. Identify the seven kinds of nodes that XPath recognizes.
46. True or false: XPath recognizes CDATA sections.
47. Describe what XPath provides for selecting nodes.
48. True or false: In a location path expression, you must prefix an attribute name with the @ symbol.
49. Identify the functions that XPath provides for selecting comment, text, and processing-instruction nodes.
50. What does XPath provide for selecting unknown nodes?
51. How do you perform multiple selections?
52. What is a predicate?
53. Identify the functions that XPath provides for working with nodesets.
54. Identify the three advanced features that XPath provides to overcome limitations with the XPath 1.0 language.
55. Define XSLT.
56. How does XSLT accomplish its work?
58. Modify books.xml to include an internal DTD that satisfies the previous exercise's requirements. Use Listing 15-10's SAXDemo application to validate books.xml against its DTD (java SAXDemo books.xml -v).
59. Create a SAXSearch application that searches books.xml for those book elements whose publisher child elements contain text that equals the application's single command-line publisher name argument. Once there is a match, output the title element's text followed by the book element's isbn attribute value. For example, java SAXSearch Apress should output title = Beginning Groovy and Grails, isbn = 9781430210450, whereas java SAXSearch "Addison Wesley" should output title = Advanced C++, isbn = 0201548550 followed by title = Effective Java, isbn = 0201310058 on separate lines. Nothing should output if the command-line publisher name argument doesn't match a publisher element's text.
60. Create a DOMSearch application that's the equivalent of the previous exercise's SAXSearch application.
61. Modify Listing 15-17's contacts document by changing <name>John Doe</name> to <Name>John Doe</Name>. Because you no longer see John Doe in the output when you run Listing 15-19's XPathSearch application (you only see Bob Jones), modify this application's location path expression so that you see John Doe followed by Bob Jones.
62. Create a books.xsl stylesheet file and a MakeHTML application with a similar structure to the application that processes Listing 15-25's recipe.xsl stylesheet. MakeHTML uses books.xsl to convert Exercise 57's books.xml content to HTML. When viewed in a web browser, the HTML should result in a web page that's similar to the page shown in Figure 15-4.

**Advanced C++**

ISBN: 0201548550
Publication Year: 1992
James O. Coplien

**Beginning Groovy and Grails**

ISBN: 9781430210450
Publication Year: 2008
Christopher M. Judd
Joseph Faisal Nusairat
James Shingler

**Effective Java**

ISBN: 0201310058
Publication Year: 2001
Joshua Bloch

*Figure 15-4. Exercise 57’s books.xml content is presented via a web page*

**Summary**

Applications often use XML documents to store and exchange data. Before you can understand these documents, you need to understand XML. This understanding requires knowledge of the XML declaration, elements and attributes, character references and CDATA sections, namespaces, and comments and processing instructions. It also involves learning what it means for a document to be well formed, and what it means for a document to be valid in terms of DTDs and XML Schema-based schemas.

You also need to learn how to process XML documents via the SAX, DOM, XMPPULL V1, XPath, and XSLT APIs. SAX is used to parse documents via a callback paradigm, DOM is used to parse documents into and create documents from node trees, XMPPULL V1 is used to parse documents in event-based contexts, XPath is used to search node trees in a more succinct manner than that offered by the DOM API, and XSLT (with help from XPath) is used to transform XML content to XML, HTML, or another format.

This chapter largely wraps up the coverage of Java APIs that are supported by Android. However, there are a few more APIs to consider. Chapter 16 presents these APIs and a few additional items.
Focusing on Odds and Ends

I’ve covered much of what you need to know about Java to give you a solid foundation on which to build with Android app fundamentals. However, there still are a few topics that you should first understand. In this chapter, I introduce you to various language, API, and miscellaneous topics (such as design patterns and the Java Native Interface) that you’ll find helpful in an Android context. I also present additional APIs (such as Preferences) that are not as useful for Android apps, but are sure to be useful for non-Android applications.

Focusing on Additional Language Features

Apache Harmony (http://en.wikipedia.org/wiki/Apache_Harmony) is the basis for the core of Android’s standard class library, which is why Android doesn’t support Java language features more recent than Java 5 and APIs more recent than Java 6. It’s also why I haven’t covered newer language features and APIs.

Caution Android doesn’t support any version of Java (including the upcoming Java 8) beyond Java 6.

Java hasn’t stopped evolving, and version 7 introduced several new language features that will be helpful to Android app developers. These features range from the small (adding underscores to integer literals, for example) to the more significant (such as automatic resource management). Unfortunately, attempts to build Android apps that leverage these features in their source codes fail. In 2012, I wrote a pair of articles for InformIT that show how to overcome various problems with JDK 7 when developing Android software:

Part 2 focuses on supporting Java 7 language features. Because you might want to follow that article’s guidelines for supporting Java 7 language features in your Android apps, and because you are probably unfamiliar with Java 7’s new language offerings, I briefly introduce you to several useful Java 7 language features in this section.

**Numeric Literal Enhancements**

Java 7 introduced two enhancements to numeric literals: binary integer literals and underscores in numeric literals.

To express an integer literal in binary notation, the literal must be prefixed with 0b or 0B and continue with 0s and 1s. For example, 0b01111111 is the binary equivalent of decimal integer 127.

To improve readability, you can insert underscores between an integer literal’s digits, for example, 204_555_1212. Although you can insert multiple successive underscores between digits (as in 0b1111__0000), you cannot specify a leading underscore (as in _123) because the compiler would treat the literal as an identifier. Also, you cannot specify a trailing underscore (as in 123_). This feature isn’t confined to integer literals. You can also make floating-point literals easier to read by placing underscores between digits (3.141_592_654, for example).

**Switch-on-String**

Another small but welcome language enhancement is switch-on-string. Starting with Java 7, you can pass a string expression to a switch statement’s selector and specify string literals for this statement’s various cases. For example, the following code fragment iterates over the array of command-line arguments passed to the main() method:

```java
public static void main(String[] args)
{
    for (int i = 0; i < args.length; i++)
        switch (args[i])
        {
            case "-v":
                case "-V": System.out.println("version 1.0");
                break;
            default : showUsage();
        }
}
```

Each loop iteration uses switch to examine the current argument to determine if it’s -v or -V. If so, a version number message is output; otherwise, the showUsage() method is invoked.
Diamond Operator

Diamonds may be a girl's (and possibly a guy's) best friend, but the diamond operator (not a true operator) is bound to be a great friend of the developer. The diamond operator is an empty pair of angle brackets (<>) that you can specify as a shorthand for the actual type arguments when instantiating a generic type. For example, consider the following verbose code fragment:

```java
Map<String, List<String>> countries = new HashMap<String, List<String>>();
```

You can use the diamond operator to replace this verbose code fragment with the following shorter code fragment that eliminates the duplicate type information:

```java
Map<String, List<String>> countries = new HashMap<>();
```

Multicatch

Java 7 introduced the ability to write a single catch block that catches more than one type of exception. This multicatch feature removes the need to catch overly broad exceptions (such as `catch (Exception ex)` to avoid code duplication. Consider the following example where multicatch isn't used:

```java
try {
   // code that may throw IOException or SQLException
} catch (IOException ioe) {
   // code that logs and otherwise handles the exception
} catch (SQLException sqle) {
   // identical code to the previous catch block's code
}
```

To eliminate the duplicate code, Java 7 lets you specify multiple exception types in the catch block header. Each successive exception type is separated from its predecessor type by placing a vertical bar (|) between these types. For example, I can refactor the previous code fragment into the following more compact code fragment:

```java
try {
   // code that may throw IOException or SQLException
} catch (IOException | SQLException iosqle) {
   // code that logs and otherwise handles exception
}
```

When either `java.io.IOException` or `java.sql.SQLException` is thrown, this common catch block handles the exception.
CHAPTER 16: Focusing on Odds and Ends

Note  When multiple exception types are listed in a catch block's header, the parameter is implicitly regarded as final.

Automatic Resource Management

Listing 11-11’s Copy application showed you how to use the java.io.FileInputStream and java.io.FileOutputStream classes to copy one file to another file. It also showed you how to close these streams and their underlying files properly, whether or not an exception was thrown. For convenience, I repeat Listing 11-11 here as Listing 16-1.

Listing 16-1. Copying a Source File to a Destination File (Version 1)

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream(args[0]);
            fos = new FileOutputStream(args[1]);
            int b; // I chose b instead of byte because byte is a reserved word.
            while ((b = fis.read()) != -1)
                fos.write(b);
        }
        catch (FileNotFoundException fnfe) {
            System.err.println(args[0] + " could not be opened for input, or "+
                                args[1] + " could not be created for output");
        }
        catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```
finally
{
    if (fis != null)
    try
    {
        fis.close();
        
    } catch (IOException ioe)
    {  assert false; // shouldn't happen in this context
    
    }
    
    if (fos != null)
    try
    {
        fos.close();
        
    } catch (IOException ioe)
    {  assert false; // shouldn't happen in this context
    
    }
    
}
}

Having to specify the boilerplate code that closes the input and output file streams is tedious and also prone to error. To overcome these problems, Java 7 introduced automatic resource management (ARM) to close open resources such as file streams automatically. It implemented ARM in the form of a new try-with-resources statement, which has the following syntax:

```
try (resource acquisition [; resource acquisition]*)
{
    // resource usage
}
```

According to this syntax, the try keyword is parameterized with a semicolon-separated list of resource-acquisition statements, where each statement acquires a resource. Each acquired resource can be accessed in the body of the try block, and it is automatically closed when execution leaves this body.

Unlike the regular try statement, try-with-resources doesn’t require the try block to be followed by catch blocks and/or a finally block. However, these blocks can be specified.

Consider the following example:

```
try (FileInputStream fis = new FileInputStream(args[0]))
{
    // do something with the connection
}
```
CHAPTER 16: Focusing on Odds and Ends

This example creates a file input stream to the file identified by args[0] before entering the try block. This stream is closed when execution leaves this block, either normally or via a thrown exception.

Listing 16-2 shows you how to use try-with-resources to simplify Listing 16-1.

Listing 16-2. Copying a Source File to a Destination File (Version 2)

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        try (FileInputStream fis = new FileInputStream(args[0]);
             FileOutputStream fos = new FileOutputStream(args[1])) {
            int b; // I chose b instead of byte because byte is a reserved word.
            while ((b = fis.read()) != -1) {
                fos.write(b);
            }
        } catch (FileNotFoundException fnfe) {
            System.err.println(args[0] + " could not be opened for input, or " +
                                args[1] + " could not be created for output");
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

The file input and file output streams are closed when execution leaves the try block, either normally or when an exception is thrown.

Compile either Listing 16-1 or Listing 16-2 as follows:

```
javac Copy.java
```

Now execute the Copy application as follows:

```
java Copy Copy.java Copy.bak
```

If all goes well, you should observe a Copy.bak file whose contents are identical to Copy.java.
Focusing on Classloaders

The virtual machine relies on classloaders to dynamically load compiled classes and other reference types (classes for short) from classfiles, Java Archive (JAR) files, URLs, and other sources into memory. Classloaders insulate the virtual machine from filesystems, networks, and so on.

In this section, I introduce you to classloaders by first presenting the various kinds of classloaders supported by Java. I then present the mechanics of loading a class. After demonstrating classloaders and revealing various difficulties with them, I close by discussing resources.

Kinds of Classloaders

When the virtual machine starts running, three classloaders are available:

- **Bootstrap** (also known as *primordial or default*) uses the operating system file I/O mechanism to load classes from the core Java libraries (such as the java.lang and java.io packages found in rt.jar), and it is written in native code.

- **Extension** loads classes from the JAR files located in the extensions directories (such as $JAVA_HOME/lib/ext). It's implemented by the sun.misc.Launcher$ExtClassLoader class (in rt.jar).

- **System** (also known as *application*) loads an application’s classes and resources found on the classpath and is implemented by the sun.misc.Launcher$AppClassLoader class (in rt.jar).

Additionally, the standard class library provides the abstract java.lang.ClassLoader class as the ultimate root class for all classloaders (including extension and system) except for bootstrap.

ClassLoader is subclassed by the concrete java.security.SecureClassLoader class, which takes security information into account; and SecureClassLoader is subclassed by the concrete java.net.URLClassLoader class, which lets you load classes and resources from a search path of URLs referring to JAR files and directories. (ExtClassLoader and AppClassLoader extend URLClassLoader.)

**Note** Android supplies additional DexClassLoader and PathClassLoader classes that extend the common BaseDexClassLoader class, which itself extends ClassLoader. These three classes exist in the dalvik.system package.

Starting with Java 1.2, classloaders have a hierarchical relationship in which each classloader except for bootstrap has a parent classloader. The bootstrap classloader is the root classloader in much the same way as java.lang.Object is the root reference type.

Along with these kinds of classloaders, Java recognizes current and context classloaders.
The current classloader is the classloader that loads the class to which the currently executing method belongs, and it is implied by methods such as java.lang.Class's `Class<?> forName(String className)` method. Behind the scenes, `forName()` relies on the current class loader to load the specified class.

When the current classloader (the bootstrap classloader is never current after loading the core Java libraries) is asked to load a class, it asks its parent to perform this task. When the parent cannot load the class, the parent asks its parent classloader to do so and this process continues on up the hierarchy. If none of these classloaders can load the class, the current classloader is given a chance. This behavior is known as delegation.

The context classloader (introduced in Java 1.2) is the classloader associated with the current thread. The current thread's context classloader is inherited from the thread's parent thread and defaults to the system classloader (which happens to be the classloader associated with the application's main—ultimate parent—thread). Also, the context classloader has a parent classloader and supports the same delegation model for class loading as previously described.

The java.lang.Thread class declares a `void setContextClassLoader(ClassLoader cl)` method that a parent thread invokes to specify a child thread's classloader before starting the child thread. A companion `ClassLoader getContextClassLoader()` method is also declared.

### Class-Loading Mechanics

Central to ClassLoader are its `Class<?> loadClass(String name)` and protected `Class<?> loadClass(String name, boolean resolve)` methods, which try to load the class with the specified name. They throw `java.lang.ClassNotFoundException` when the class cannot be found or return a `Class` object representing the loaded class. The former method invokes the latter method passing `false` to `resolve` (In the Android reference implementation, `loadClass(String, boolean)`'s second parameter is ignored.)

**Note** The java.lang.String object passed to `name` must specify the class's binary name, which adheres to the convention that's specified in the Java Language Specification (http://docs.oracle.com/javase/specs/). Examples include "java.lang.String" and "java.net.URLClassLoader$3$1".

The `loadClass(String name, boolean resolve)` method performs the following tasks.

1. It invokes ClassLoader's protected final `Class<?> findLoadedClass(String name)` method to return the class with the given binary name when the calling classloader has been recorded by the virtual machine as an initiating loader of a class with that binary name. Otherwise, null is returned. This method returns the class from the calling classloader's class cache when the class was previously loaded (and is in this cache for performance reasons).

2. When `findLoadedClass()` returns null, `loadClass(String, boolean)` invokes `loadClass(String, boolean)` on the non-null parent classloader or invokes an internal method requesting that the bootstrap classloader handle this task.
3. When neither the parent nor any of its parents (including bootstrap) locates the class, ClassNot FoundException is thrown and the initial loadClass(String, boolean) method call invokes ClassLoader's protected Class<?> findClass(String name) method to find and load the class.

4. The findClass() method locates the class or throws ClassNot FoundException when the class cannot be found; this exception is thrown out of loadClass(String, boolean) and loadClass(String). (ClassLoader's findClass() method always throws ClassNot FoundException.)

5. Assuming that the class is located, findClass() loads the class's compiled representation into an array of bytes. It then (ultimately) invokes one of ClassLoader's defineClass() methods to convert these bytes into a Class object but only after defineClass() runs these bytes through the bytecode verifier (see Chapter 1) to ensure that they don't compromise virtual machine security.

6. Assuming that all is well, findClass() returns a Class object. When true is passed to loadClass(String, boolean)'s resolve parameter, ClassLoader's protected final void resolveClass(Class<?> c) method is called to resolve the class. Resolution causes any other classes that are immediately referenced from the loaded class (such as a static variable of another class type) to be loaded by this classloader, classes immediately referenced from those classes to be loaded, and so on. (Classes used as instance variables, parameters, or local variables are not normally loaded at this time. They're loaded when actually referenced by the class.)

findLoadedClass() is declared protected so that it can be accessed by subclasses in different packages. This method is declared final so that it cannot be overridden. The same is true for resolveClass().

**Tip** When you need your own classloader, you should first consider using URLClassLoader, which will save you a lot of work, and which leverages the security features provided by its SecureClassLoader parent. If you prefer to subclass ClassLoader directly, you'll minimally need to override findClass().

**Playing with Classloaders**

I've created a pair of applications to help you start to explore classloaders. Listing 16-3 presents the first application, which reveals that the main thread's context classloader is the system classloader.
Listing 16-3. Proving that the Main Thread's Context Classloader is the System Classloader

```java
public class ClassLoaderDemo {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getContextClassLoader());
        System.out.println(ClassLoader.getSystemClassLoader());
    }
}
```

Listing 16-3's main() method first obtains the main thread's context classloader and outputs this classloader's name. This method then invokes ClassLoader's ClassLoader getSystemClassLoader() class method, which returns a reference to the system classloader's ClassLoader object. The name of this classloader is then output.

Compile Listing 16-3 as follows:

```
javac ClassLoaderDemo.java
```

Run the application as follows:

```
java ClassLoaderDemo
```

You should observe output similar to that shown here:

```
sun.misc.Launcher$AppClassLoader@26e2e276
sun.misc.Launcher$AppClassLoader@26e2e276
```

My second application demonstrates URLClassLoader. It uses this classloader to load a class named Hello via a URL. This class's Hello.class classfile is located in directory x, which is a subdirectory of the current directory (the directory in which the virtual machine is launched via the java tool). Listing 16-4 presents Hello.java.

Listing 16-4. A Simple Demonstration Class

```java
public class Hello {
    static {
        System.out.println("Welcome to Hello");
    }

    static int x = 1;

    public static void main(String[] args) {
        System.out.println("Hello");
        System.out.println("Number of arguments = " + args.length);
        System.out.println("x = " + x);
    }
}
```
Hello must be declared public; otherwise, URLClassLoader outputs an error message about its not being able to access this class.

Assuming that Hello.java is stored in x, a subdirectory of the current directory, compile Listing 16-4, as follows:

javac x/Hello.java

Listing 16-5 presents an application that loads and runs this class.

Listing 16-5. Attempting to Load Hello.class via a File-Based URL

```java
import java.lang.reflect.Method;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassLoaderDemo
{
    final static String _URL_ = "file:///" + System.getProperty("user.dir") + "/x/;"

    public static void main(String[] args)
    {
        boolean init = true;
        if (args.length == 1 && args[0].equalsIgnoreCase("noinit"))
            init = false;
        try
        {
            URL[] urls = new URL[] { new URL(_URL_) };
            URLClassLoader urlc = new URLClassLoader(urls);
            Class<?> clazz = Class.forName("Hello", init, urlc);
            System.out.println(clazz.getClassLoader());
            run(clazz);
        }
        catch (ClassNotFoundException cnfe)
        {
            System.err.println("Class not found");
        }
        catch (MalformedURLException murle)
        {
            System.err.println("URL is malformed");
        }
    }

    static void run(Class<?> clazz)
    {
        try
        {
            Method main = clazz.getMethod("main", new Class[] { String[].class });
            Object[] args = new Object[] { new String[0] };
            main.invoke(null, args);
        }
    }
```
catch (Exception e) {
    System.err.println(e.getMessage());
}
}

Listing 16-5’s `main()` method first examines its array of command-line arguments for the presence of a `noinit` argument and resets the `init` variable (which is initially true) to false when this argument is specified. (I’ll explain this variable’s purpose shortly.)

`main()` next attempts to instantiate `URLClassLoader` by passing an array of one `java.net.URL` instance to this class’s constructor. The string-based URL argument passed to URL’s constructor consists of the “file://” protocol prefix (to access the local filesystem), followed by `System.getProperty("user.dir")`’s result (the current working directory), followed by “/x/”. (The final “/” is required; otherwise, `URLClassLoader` assumes that `x` is the name of a JAR file. Because this JAR file doesn’t exist and obviously doesn’t contain `Hello.class`, `ClassNotFoundException` is thrown.)

Assuming that `URLClassLoader` is successfully created, `main()` invokes `Class`'s `Class<?>.forName(String name, boolean initialize, ClassLoader loader)` method to try to load the class identified by `name` and using `loader`. Passing true to initialize causes the loaded class to be statically initialized. Otherwise, the loaded class isn’t statically initialized. The `init` variable lets you explore both initialization scenarios.

The returned `Class` object’s `ClassLoader getClassLoader()` method is invoked to return the `ClassLoader` instance used to load the class. This name is subsequently output.

`ClassLoaderDemo`'s `void run(Class<?> clazz)` class method is now called to instantiate the loaded class and invoke its `main()` method with help from the Reflection API (discussed in Chapter 8).

Assuming that `ClassLoaderDemo.java` is located with `x` in the current directory, compile this source file (`javac ClassLoaderDemo.java`) and run the application (`java ClassLoaderDemo`). The application responds by presenting the following output (except possibly for `56092666`):

```
Welcome to Hello
java.net.URLClassLoader@56092666
Hello
Number of arguments = 0
x = 1

```

This output order proves that `Class.forName()` statically initialized `Hello` (true was passed to `initialize`) by invoking its `<clinit>()` virtual machine method (see Chapter 3).

Continue by running the application as follows:

```
java ClassLoaderDemo noinit
```

www.it-ebooks.info
The application responds by presenting the following output (except possibly for 56092666):

```
java.net.URLClassLoader@56092666
Welcome to Hello
Hello
Number of arguments = 0
x = 1
```

This output order proves that `Class.forName()` didn’t statically initialize `Hello`. However, this class is statically initialized just before `run()` launches its `main()` method.

Copy `Hello.class` into the same directory as `ClassLoaderDemo.class`, and then execute `java ClassLoader`. This time you’ll observe the following output (except possibly for 26e2e276):

```
Welcome to Hello
sun.misc.Launcher$AppClassLoader@26e2e276
Hello
Number of arguments = 0
x = 1
```

The reason for this different output is that `URLClassLoader` delegates to its parent classloader, which happens to be the system classloader. The system classloader locates `Hello.class` in the current directory and causes `Class.forName()` to return this class’s `Class` object.

### Classloader Difficulties

Context classloaders can be a source of difficulty as discussed in “Find a way out of the ClassLoader maze” ([www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.html](www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.html)), a JavaWorld article by Vladimir Roubtsov. I’ve created a simple example that demonstrates this difficulty.

My example is based on two versions of a class named `Version`. Listing 16-6 presents the first version’s source code.

**Listing 16-6. A Simple Demonstration Class**

```java
public class Version {
    public static void main(String[] args) {
        System.out.println("Version 1");
    }
}
```

---

www.it-ebooks.info
Assuming that Version.java is located in x, a subdirectory of the current directory, compile Listing 16-6 as follows:

```
javac x/Version.java
```

Create a y subdirectory of the current directory and copy a modified version of Listing 16-6 (replace Version 1 with Version 2) to y. Then compile this source code as follows:

```
javac y/Version.java
```

Listing 16-7 presents a ClassLoaderDemo application that demonstrates a conflict between these versions.

**Listing 16-7. Entering Classloader Hell**

```java
import java.lang.reflect.Method;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassLoaderDemo
{
    final static String CD = System.getProperty("user.dir");
    final static String _URL1_ = "file:///" + CD + "/x/";
    final static String _URL2_ = "file:///" + CD + "/y/";

    public static void main(String[] args)
    {
        try
        {
            URL[] urls = new URL[] { new URL(_URL1_ )};
            URLClassLoader urlc1 = new URLClassLoader(urls);
            Class<?> clazz1 = Class.forName("Version", true, urlc1);
            run(clazz1);
            urls = new URL[] { new URL(_URL2_) };
            URLClassLoader urlc2 = new URLClassLoader(urls);
            Class<?> clazz2 = Class.forName("Version", true, urlc2);
            run(clazz2);
            Thread.currentThread().setContextClassLoader(urlc1);
            run(Thread.currentThread().getContextClassLoader().loadClass("Version"));
            Thread.currentThread().setContextClassLoader(ClassLoader.getSystemClassLoader());
            run(Thread.currentThread().getContextClassLoader().loadClass("Version"));
        }
        catch (ClassNotFoundException cnfe)
        {
            System.err.println("Class not found");
        }
    }
}
```
catch (MalformedURLException murle)
{
    System.err.println("URL is malformed");
}

static void run(Class<?> clazz)
{
    try
    {
        Method main = clazz.getMethod("main", new Class[]{ String[].class });
        Object[] args = new Object[] { new String[0] };
        main.invoke(null, args);
    }
    catch (Exception e)
    {
        System.err.println(e.getMessage());
    }
}

Listing 16-7 loads both Version classes; each classloader caches its own version. It then sets the context classloader, first to the URLClassLoader instance and then to the system classloader. Each time, it subsequently calls loadClass(String).

Compile Listing 16-7 (javac ClassLoaderDemo.java) and run the application (java ClassLoaderDemo). The following output is generated:

Version 1
Version 2
Version 1
Class not found

The system classloader doesn't include Version in its cache because it didn't load Version. As a result, loadClass(String) throws ClassNotFoundException.

Although this example is contrived, it illustrates problems that can occur when you're not aware of which classloader is the context classloader. You'll either observe a thrown ClassNotFoundException instance or you may end up with the wrong version of a class.

Classloaders and Resources

Classloaders are typically used to load classes, but they can also load arbitrary resources (such as images) via ClassLoader methods such as InputStream getResourceAsStream(String name). Although you could call these methods directly, it's common practice to work with Class's URL getResource(String name) and InputStream getResourceAsStream(String name) methods instead. These methods differ in that getResourceAsStream() ultimately invokes getResource() and then invokes URL's InputStream openStream() method on the resulting URL instance to return an input stream.
I’ve created an application that demonstrates Class’s getResourceAsStream() method. Listing 16-8 presents its source code.

Listing 16-8. Loading an Image Resource and Viewing a Prefix of Its Content

```java
import java.io.File;
import java.io.InputStream;
import java.io.IOException;

public class ClassLoaderDemo
{
    final static String IMAGE = "mars.jpg";

    public static void main(String[] args)
    {
        System.out.println(ClassLoaderDemo.class.getClassLoader());
        InputStream is = ClassLoaderDemo.class.getResourceAsStream(IMAGE);
        if (is == null)
        {
            System.err.printf("%s not found%n", IMAGE);
            return;
        }
        try
        {
            byte[] image = new byte[(int) new File(IMAGE).length()];
            int _byte, i = 0;
            while ((_byte = is.read()) != -1)
            {
                image[i++] = (byte) _byte;
                for (i = 0; i < 16; i++)
                    System.out.printf("%02X ", image[i]);
            }
        } catch (IOException ioe)
        {
            System.err.println("I/O error: "+ ioe.getMessage());
        }
    }
}
```

Listing 16-8 specifies expression ClassLoaderDemo.class to obtain the Class object returned from the classloader that loaded ClassLoaderDemo.class, and then invokes getResourceAsStream(IMAGE) on the returned Class object. This image needs to be located in the current directory.

Assuming that this image resource can be found (getResourceAsStream() doesn’t return null), a byte array is allocated to hold the array and the image’s bytes are read over the input stream and stored in the byte array. The first 16 bytes from this array are then sent to standard output.

Compile Listing 16-8 (javac ClassLoaderDemo.java) and run the application (java ClassLoaderDemo). You should observe the following output:

```
sun.misc.Launcher$AppClassLoader@26e2e276
FF D8 FF E0 00 10 4A 46 49 46 00 01 02 01 00 48
```
It's common to store resources in JAR files and then access them via `getResourceAsStream()`. For example, execute the following command (which assumes that the current directory contains `mars.jpg`) to create an `image.jar` file containing `mars.jpg`:

```
jar cf image.jar mars.jpg
```

After creating this file, erase `mars.jpg` from the current directory. Continue by executing the following command:

```
java -cp image.jar;. ClassLoaderDemo
```

Note that `image.jar` and the current directory must be on the classpath so that the system classloader can load classes and resources as necessary. This time, you'll observe a thrown `java.lang.ArrayIndexOutOfBoundsException` instance. This exception occurs because you're trying to populate a zero-length byte array. This array has zero length because `(int) new File(IMAGE).length()` returns 0 (`mars.jpg` cannot be found—you just erased it).

One way to solve this problem is to remove `(int) new File(IMAGE).length()` and hardcode the image's length instead. Although this refactoring works, it isn't a good idea because it can result in a runtime exception should the size of the image change.

A better solution is to calculate the length of the file by reading the input stream and then obtain a new input stream for reading file content. Listing 16-9 demonstrates this technique.

Listing 16-9. Loading an Image Resource in a More Robust Manner and Viewing a Prefix of Its Content

```
import java.io.File;
import java.io.InputStream;
import java.io.IOException;

public class ClassLoaderDemo
{
    final static String IMAGE = "mars.jpg";

    public static void main(String[] args)
    {
        System.out.println(ClassLoaderDemo.class.getClassLoader());
        InputStream is = ClassLoaderDemo.class.getResourceAsStream(IMAGE);
        if (is == null)
        {
            System.err.printf("%s not found\n", IMAGE);
            return;
        }
        try
        {
            byte[] image = new byte[getLength(is)];
            is = ClassLoaderDemo.class.getResourceAsStream(IMAGE);
            int _byte, i = 0;
```

www.it-ebooks.info
while ((_byte = is.read()) != -1)
    image[i++] = (byte) _byte;
for (i = 0; i < 16; i++)
    System.out.printf("%02X ", image[i]);
}
catch (IOException ioe)
{
    System.err.println("I/O error: " + ioe.getMessage());
}
}

static int getLength(InputStream is) throws IOException
{
    byte[] buffer = new byte[1024];
    int length = 0;
    int bytesRead;
    while ((bytesRead = is.read(buffer)) > 0)
    {
        length += bytesRead;
    }
    return length;
}

Listing 16-9 introduces a `getLength(InputStream)` class method that reads the contents of the input stream into a fixed-size buffer in multiple steps. The number of bytes read is accumulated in a length variable whose value is returned from the method.

This time, ClassLoaderDemo will successfully read the contents of the image whether or not the image is stored in a JAR file.

**Focusing on Console**

You’re writing a console-based application that runs on the server. This application needs to prompt the user for a username and a password before granting access. Obviously, you don’t want the password to be echoed to the console (such as a command window).

Before Java 6, you had almost no way to accomplish this task without resorting to the Java Native Interface (discussed later). Although `java.awt.TextField` provides a `void setEchoChar(char c)` method for this purpose, this method is only appropriate for GUI-based/non-Android applications.

Note  AWT stands for *Abstract Window Toolkit*, a windowing toolkit that makes it possible to create crude user interfaces consisting of windows, buttons, text fields (via the `TextField` class), and so on. The AWT was released as part of Java 1.0 in 1995, and it continues to be part of Java’s standard class library. The AWT isn’t supported by Android.

Java 6 responded to this need by introducing the `java.io.Console` class. This class declares methods that access the platform’s character-based console device but only when that device is associated with the current virtual machine.
CHAPTER 16: Focusing on Odds and Ends

Table 16-1. Console Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void flush()</td>
<td>Flushes the console, immediately writing any buffered output.</td>
</tr>
<tr>
<td>Console format(String fmt, Object... args)</td>
<td>Writes a formatted string to the console's output stream. The Console reference is returned so that you can chain method calls together (for convenience). This method throws java.util.IllegalArgumentException when the format string contains illegal syntax. (I discussed format strings in the context of the java.util.Formatter class in Chapter 13.)</td>
</tr>
<tr>
<td>Console printf(String format, Object... args)</td>
<td>An alias for format().</td>
</tr>
<tr>
<td>Reader reader()</td>
<td>Returns the java.io.Reader instance associated with the console. This instance can be passed to a java.util.Scanner constructor for sophisticated scanning/parsing. (I discussed Scanner in Chapter 13.)</td>
</tr>
<tr>
<td>String readLine()</td>
<td>Reads a single line of text from the console's input stream. The line (minus line-termination characters) is returned in a String object. This method returns null when the end of the stream is reached. It throws java.io.IOException when an I/O error occurs.</td>
</tr>
</tbody>
</table>

Note Whether or not a virtual machine has a console is dependent upon the underlying platform and also upon the manner in which the virtual machine is invoked. When the virtual machine is started from an interactive command line without redirecting the standard input and output streams, its console will exist and (typically) will be connected to the keyboard and display from which the virtual machine was launched. When the virtual machine is started automatically (such as by a background job scheduler), it usually won’t have a console.

To determine if a console is available, call the java.lang.System class’s Console console() class method, as follows:

```java
Console console = System.console();
if (console == null)
{
    System.err.println("no console device is present");
    return;
}
```

This method returns a Console reference when a console is present; otherwise, it returns null. After verifying that null wasn’t returned, you can use the reference to call Console’s methods (see Table 16-1).
Table 16-1. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String readLine(String fmt, Object... args)</code></td>
<td>Writes a formatted prompt string to the console's output stream and then reads a single line of text from its input stream. The line (minus line-termination characters) is returned in a <code>String</code> object. This method returns <code>null</code> when the end of the stream has been reached. It throws <code>IllegalFormatException</code> when the format string contains illegal syntax and <code>IOError</code> when an I/O error occurs.</td>
</tr>
<tr>
<td><code>char[] readPassword()</code></td>
<td>Reads a password or passphrase from the console's input stream with echoing disabled. The password/passphrase (minus line-termination characters) is returned in a <code>char</code> array. This method returns <code>null</code> when the end of the stream has been reached. It throws <code>IOError</code> when an I/O error occurs.</td>
</tr>
<tr>
<td><code>char[] readPassword(String fmt, Object... args)</code></td>
<td>Writes a formatted prompt string to the console's output stream and then reads a password/passphrase from its input stream with echoing disabled. The password/passphrase (minus line-termination characters) is returned in a <code>char</code> array. This method returns <code>null</code> when the end of the stream has been reached. It throws <code>IllegalFormatException</code> when the format string contains illegal syntax and <code>IOError</code> when an I/O error occurs.</td>
</tr>
<tr>
<td><code>PrintWriter writer()</code></td>
<td>Returns the unique <code>java.io.PrintWriter</code> instance associated with this console.</td>
</tr>
</tbody>
</table>

I've created a `Login` application that invokes `Console` methods to obtain a username and a password. Listing 16-10 presents `Login`'s source code.

Listing 16-10. Simulating a Username/Password Login Operation

```java
import java.io.Console;
import java.io.IOException;

public class Login {
    public static void main(String[] args) {
        Console console = System.console();
        if (console == null) {
            System.err.println("no console device is present");
            return;
        }
        try {
            String username = console.readLine("Username:");
            char[] pwd = console.readPassword("Password:");
            // Do something useful with the username and password. For something
            // to do, this application just prints out these values.
            System.out.println("Username = " + username);
            System.out.println("Password = " + new String(pwd));
        }
    }
}
```
// Prepare username String for garbage collection. More importantly,
// destroy the password.
username = "";
for (int i = 0; i < pwd.length; i++)
    pwd[i] = 0;
}
catch (IOException ioe)
{
    console.printf("I/O error: %s\n", ioe.getMessage());
}
}

Compile Listing 16-10 as follows:

javadoc Login.java

Run this application as follows:

java Login

The application first prompts you to enter a username that's displayed and then prompts for a
password that isn’t displayed. After entering the password, messages identifying the username and
password are written to standard output.

Note After obtaining and (presumably) doing something useful with the entered username and password,
it’s important to get rid of these items for security reasons. Most importantly, you’ll want to remove the
password by zeroing out the char array.

Focusing on Design Patterns

Designing significant applications is often difficult and prone to error. For example, a poorly designed
application might suffer from the fragile base class problem that I discussed in Chapter 4. Over the
years, developers have encountered various design problems and have devised clever solutions.
These problems and their solutions have been catalogued to help other developers detect them and
avoid reinventing solutions. These catalogued entities are known as design patterns.

A design pattern is a catalogued problem/solution entity that consists of four components:

- Name: Each design pattern has a name that describes the pattern and provides
  a vocabulary for discussing it with other developers.
- Problem: Each design pattern clearly states the problem that it solves and the
  context in which the problem occurs. It tells you when to apply the design pattern.
Solution: Each design pattern identifies the classes and objects along with their relationships and other factors that solve the problem.

Consequences: Choosing one design pattern over another pattern involves trade-offs that can impact your application's flexibility and future maintenance.

Perhaps the most famous design patterns catalog is Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; ISBN: 0201633612). This book presents 23 design patterns, such as the Decorator pattern that I briefly mentioned in Chapter 4 for solving the fragile base class problem. It uses the C++ programming language to codify these patterns.

To give you an appetite for design patterns, which you might find helpful when designing Android apps, I present the Strategy pattern in a Java context.

Understanding Strategy

The Strategy pattern lets you define a family of algorithms (such as sorting algorithms), encapsulate each algorithm in its own class, and make these algorithms interchangeable. Each encapsulated algorithm is known as a strategy. At runtime, your application chooses the appropriate algorithm that meets its requirements.

Unlike Decorator, which lets you change an object’s appearance, Strategy lets you change an object’s behaviors. You move conditional branches into their own strategy classes to avoid multiple conditional statements. These classes often implement a Java interface or derive from an abstract superclass, which your application references and uses to interact with a specific strategy.

There are three participants in the Strategy pattern:

- **Strategy**: A common interface to all supported algorithms.
- **ConcreteStrategy**: An implementation of the Strategy interface for a specific algorithm. Multiple ConcreteStrategy participants are common.
- **Context**: The context in which the concrete strategy is invoked.

Implementing Strategy

Consider an abstract implementation consisting of Strategy, ConcreteStrategyx (x is an integer that uniquely identifies a strategy), and Context types. Listing 16-11 presents Strategy.


```java
public interface Strategy {
    public void execute(String msg);
}
```

Listing 16-11’s Strategy interface offers an abstract execute() method that must be implemented by each class that implements this interface. This method executes a strategy algorithm.
In this example, `execute()` returns nothing, although it could be declared to return a value of a specific type. Also, `execute()` declares a single `String` parameter. `execute()`’s return type and parameter list depend upon the family of algorithms that are implemented.

You typically choose a Java interface to implement the common strategy interface. However, you can also use an abstract class, especially when you need to store data that’s common to all concrete strategies, for example, the width in which to align text for left-alignment, right-alignment, and justify text-alignment strategies.

Listing 16-12 presents a concrete strategy implementation.

**Listing 16-12. ConcreteStrategy1 Provides an Implementation of the Strategy Interface**

```java
public class ConcreteStrategy1 implements Strategy {
    @Override
    public void execute(String msg) {
        System.out.printf("executing strategy #1: msg = %s\n", msg);
    }
}
```

Listing 16-13 presents a second concrete strategy implementation.

**Listing 16-13. ConcreteStrategy2 Provides a Second Implementation of the Strategy Interface**

```java
public class ConcreteStrategy2 implements Strategy {
    @Override
    public void execute(String msg) {
        for (int i = 0; i < 3; i++)
            System.out.printf("executing strategy #2: msg = %s\n", msg);
    }
}
```

In each of Listings 16-12 and 16-13, you can think of `execute()` as performing a more useful task such as executing a specific sorting algorithm (Quick Sort or Bubble Sort, for example).

Listing 16-14 presents a Context class for invoking concrete strategies.

**Listing 16-14. Context Provides a Framework for Invoking Concrete Strategy Implementations**

```java
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        setStrategy(strategy);
    }
```
public void executeStrategy(String msg) {
    strategy.execute(msg);
}

public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
}
}

Listing 16-14's Context class stores a concrete strategy when created, provides a method to subsequently change the strategy, and provides another method to execute the current strategy.

Finally, Listing 16-15 presents a StrategyDemo application that demonstrates the strategy pattern via these types.


public class StrategyDemo {
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStrategy1());
        context.executeStrategy("Hello");
        context.setStrategy(new ConcreteStrategy2());
        context.executeStrategy("World");
    }
}

Context is instantiated and its instance is configured to the first concrete strategy. The Context object then executes this strategy with a Hello text message. Next, the Context object is reconfigured with the second concrete strategy and then executes this strategy with a World text message.

Compile Listing 16-15 as follows:

javac StrategyDemo.java

Assuming that all source files are located in the same current directory, compilation should succeed.

Execute StrategyDemo as follows:

java StrategyDemo

You should observe the following output:

compiling strategy #1: msg = Hello
executing strategy #2: msg = World
executing strategy #2: msg = World
executing strategy #2: msg = World
executing strategy #2: msg = World
CHAPTER 16: Focusing on Odds and Ends

Note  Check out Pankaj Kumar's “Strategy Design Pattern in Java – Example Tutorial” blog post (www.javacodegeeks.com/2013/08/strategy-design-pattern-in-java-example-tutorial.html) for a more useful shopping cart example of the strategy design pattern.

Focusing on Double Brace Initialization

While discussing anonymous classes in Chapter 5, I stated the following:

Although an anonymous class doesn’t have a constructor, you can provide an instance initializer to handle complex initialization. For example, new Office() {{addEmployee(new Employee("John Doe"));}}; instantiates an anonymous subclass of Office and adds one Employee object to this instance by calling Office’s addEmployee() method.

The previous code fragment can be expressed less compactly but perhaps more clearly as follows:

```java
new Office()
{
    { addEmployee(new Employee("John Doe");
    }
};
```

The add-employee code within the instance initializer is executed when an object is created from an anonymous subclass of the Office class.

This example demonstrates what is commonly referred to as double brace initialization (www.c2.com/cgi/wiki?DoubleBraceInitialization). Although convenient, double brace initialization has a couple of drawbacks that you need to understand.

One drawback is a bloated number of classfiles (http://stackoverflow.com/questions/924285/efficiency-of-java-double-brace-initialization). Anonymous classes contribute to the number of classfiles comprising an application, which can greatly increase its size. This increased size can be especially problematic when installing and running Android apps on devices with limited storage/memory space.

A second drawback is that you are unable to use Java 7’s diamond operator (http://java.dzone.com/articles/double-brace-initialization), which you might want to use after reading my article (mentioned earlier in this chapter) on supporting Java 7 language features. You cannot specify the diamond operator because the compiler cannot infer type arguments. For example, consider the following code fragment:

```java
Map<String, String> capitals = new HashMap<>(){
    {
        put("Canada", "Ottawa");
        put("England", "London");
        put("France", "Paris");
    }
};
```
The compiler reports an error when it encounters the diamond operator. The corrected code fragment, which specifies the type arguments when instantiating `HashMap`, appears below:

```java
Map<String, String> capitals = new HashMap<String, String>() {
    {
        put("Canada", "Ottawa");
        put("England", "London");
        put("France", "Paris");
    }
};
```

**Focusing on Fluent Interfaces**

While discussing methods in Chapter 3, I referred to instance method call chaining in which instance method calls are chained together as in `new SavingsAccount().deposit(1000).printBalance();`.

Instance method call chaining is more compactly known as *method chaining*. Each method call in the chain returns an object, allowing the calls to be chained together in a single statement. Chaining is syntactic sugar that eliminates the need for intermediate variables.

Method chaining is used in the construction of *fluent interfaces*, which are implementations of object-oriented APIs that provide for more readable code.

According to Wikipedia’s “Fluent interface” topic ([http://en.wikipedia.org/wiki/Fluent_interface](http://en.wikipedia.org/wiki/Fluent_interface)), a fluent interface is normally implemented via method chaining to relay the instruction context of a subsequent method call. Furthermore, the context is defined through the return value of a called method, the context is self-referential in that the new context is equivalent to the last context, and the context is terminated through the return of a void context.

The Java Object Oriented Querying library (commonly known as jOOQ) is a light database-mapping software library that offers a well-known example of a fluent interface. For example, consider the following SQL query for selecting authors with sold-out books:

```
SELECT * FROM AUTHOR a
WHERE EXISTS (SELECT 1
    FROM BOOK
    WHERE BOOK.STATUS = 'SOLD OUT'
    AND BOOK.AUTHOR_ID = a.ID);
```

This query can be expressed compactly in Java via the jOOQ fluent interface:

```java
create.selectFrom(AUTHOR.as("a"))
    .where(exists(selectOne()
        .from(BOOK)
        .where(BOOK.STATUS.equal(BOOK_STATUS.SOLD_OUT))
        .and(BOOK.AUTHOR_ID.equal(a.ID))));
```

Notice that method calls typically have short names (such as `where`, `from`, and `and`). Also, method chaining is used to connect these method calls together, the return value from each method call returns the same self-referential context, and the context returned from `create` is most likely void.
If you're interested in creating your own fluent interfaces, I invite you to check out Neal Ford’s “Evolutionary architecture and emergent design: Fluent interfaces” article (www.ibm.com/developerworks/java/library/j-eaed14/index.html).

Note Perhaps you’re familiar with the concept of a builder and want to know how it compares to a fluent interface. For the details, check out stackoverflow.com's “What is the difference between fluent interface and builder pattern?” topic (http://stackoverflow.com/questions/17937755/what-is-the-difference-between-fluent-interface-and-builder-pattern).

Focusing on Immutability

Earlier in this book, I introduced you to the final keyword. To recap, you use final in the following contexts:

- Declare a true constant in a class. Example: `final static PI = 3.14159;`
- Declare a blank final in a class. Unlike a true constant, a blank final is only a constant within the context of an object. Example: `final int ID;`
- Prevent a class from being subclassed to avoid the fragile base class problem. Example: `final class SavingsAccount`
- Prevent a method from being overridden for security or other reasons. Example: `final void deposit(BigDecimal amount)`
- Access a local variable or parameter from within an anonymous or local class. Example: `public static void main(final String[] args)`

Tip Some developers prefer to declare all local and parameter variables that will never change after being initialized final to prevent bugs that arise from subsequently assigning values to these variables by accident. The compiler would detect such assignment attempts and report errors.

The final keyword plays a large role in the creation of immutable classes, which are classes whose instances cannot be modified. The String and java.lang.Boolean classes are examples.

Immutable classes offer several advantages, which is why you might consider designing most of your classes to be immutable:

- Objects created from immutable classes are thread-safe and there are no synchronization issues. Because they cannot be corrupted when used by multiple threads, you can freely share them among threads. To encourage reuse of these shared objects (and to reduce garbage collection), you often find, in various immutable classes, precreated instances of common values that are expressed as public static final constants. For example, Boolean declares TRUE and FALSE Boolean constants that represent the common true and false values.
You can share the internals of an immutable class to reduce memory usage and improve performance. For example, the String class provides an internal array of characters named value. When you invoke a substring() method, a new String object is created that references the same value array as the String object on which this method was called. However, a different offset and length are passed to the object so that only the portion of the parent string delimited by the offset and length is observed as the substring. The result is a memory savings and reduced copying to improve performance.

Immutable classes support failure atomicity (a term coined by Joshua Bloch in his Effective Java books), which means that, after throwing an exception, an object is still in a well-defined and usable state, even when the exception occurred during an operation.

Immutable classes are excellent building blocks for more complex classes. The fact that immutable objects are thread-safe makes it easier to design complex classes that maintain their invariants.

Immutable objects make good java.util.Map keys and java.util.Set elements because objects must not change state while in a collection.

Several guidelines must be followed to ensure that a class is immutable:

Don’t include setter or other mutator methods in the class design. Instances of the class must never be modified following initialization.

Prevent methods from being overridden. Doing so prevents a subclass from overriding a method and attempting to use it to break immutability. You can easily accomplish this task by declaring the class final.

Declare all fields final. You should also declare those fields that are part of the implementation private so that you are free to change the implementation without breaking clients.

Prevent the class from exposing any mutable state. Clients should never be able to access fields that reference mutable objects. Also, never directly assign a client-provided object reference to a field or return an object reference from a getter or other accessor method. Instead, make defensive copies in constructors, accessor methods, and the readObject() method (when using serialization).

Making a defensive copy isn’t a difficult task. For example, the following code fragment shows you how to make a defensive copy of a mutable and fictitious Date object:

```java
Date getHireDate()
{
    return new Date(hireDate.getValue());
}
```

This code fragment assumes a private hireDate field that stores the date on which an employee was hired. This field is of type Date, which provides a getValue() method for returning the date’s value. Additional particulars of the Date class aren’t important. Instead, note that you are constructing a new Date object that is equivalent to the Date object referenced from the hireDate field. The internal Date object cannot be modified by external code that invokes getHireDate().
You would also make defensive copies of arrays. For example, suppose that you have an internal grades array of type int[] and you are declaring a getGrades() method to return this array. The following code fragment shows you how to create and return a copy of this array so that the original array cannot be modified by external code:

```java
int[] getGrades()
{
    int[] copy = new int[grades.length];
    System.arraycopy(grades, 0, copy, 0, copy.length);
    return copy;
}
```

The System.arraycopy() method is used to quickly copy the original grades array to the new copy array.

If your array contains references to mutable objects, it's not enough to make a copy of the array: you also need to make a copy of each array element, as follows:

```java
Date[] getDates()
{
    Date[] copy = new Date[dates.length];
    System.arraycopy(dates, 0, copy, 0, copy.length);
    for (int i = 0; i < dates.length; i++)
        copy[i] = new Date(dates[i].getValue());
    return copy;
}
```

Listing 16-16 presents an example of an immutable Recipe class that follows the previous guidelines. For brevity, I've omitted import statements along with the Ingredient and Step classes.

**Listing 16-16. An Immutable Recipe Class**

```java
public final class Recipe
{
    private String name;
    private List<Ingredient> ingredients;
    private List<Step> steps;

    public Recipe(String name, Ingredient[] ingredients, Step[] steps)
    {
        this.name = name;
        ingredients = new ArrayList<Ingredient>();
        for (Ingredient ingredient: ingredients)
            ingredients.add(ingredient);
        steps = new ArrayList<Step>();
        for (Step step: steps)
            steps.add(step);
    }

    public List<Ingredient> getIngredients()
    {
        return new ArrayList<Ingredient>(ingredients);
    }
}
```

www.it-ebooks.info
public String getName()
{
    return name;
}

public List<Step> getSteps()
{
    return new ArrayList<Step>(steps);
}

Notice that the constructor and getIngredients()/getSteps() getters make defensive copies of the ingredients/steps arrays and array lists, respectively. In contrast, it isn't necessary to make a defensive copy of the name parameter in the constructor and name field in the getName() getter because name is of type String, which is immutable.

**Note**  Before you can use Recipe in a hashmap context, you also have to override the equals() and hashCode() methods.

---

**Focusing on Internationalization**

We tend to write software that reflects our cultural backgrounds. For example, a Spanish developer's application might present Spanish text, an Arabic developer's application might present a Hijri (Islamic) calendar, and a Japanese developer's application might display its currencies using the Japanese Yen currency symbol. Because cultural issues restrict the size of an application's audience, you might consider internationalizing your applications to reach a larger audience (and make more money).

*Internationalization* is the process of creating an application that automatically adapts to its current user's culture (without recompilation) so that the user can read text in the user's language and otherwise interact with the application without observing cultural issues. Java simplifies internationalization by supporting *Unicode* (a universal character set that encodes the various symbols making up the world's written languages) via the char keyword (see Chapter 2) and the java.lang.Character class (see Chapter 7), and by offering the APIs discussed in this section.

Related to internationalization is the concept of *localization*, which is the adaptation of internationalized software to support a new culture by adding culture-specific elements (such as text strings that have been translated to the culture). Java already provides much of this support via various APIs, and it also lets you extend its support via locale-sensitive services, which are not discussed for brevity. Because Android differs somewhat where localization is concerned, I'll have more to say about this topic in Appendix C.
Locales

The `java.util.Locale` class is the centerpiece of the various Internationalization APIs. Instances of this class represent *locales*, which are geographical, political, or cultural regions.

`Locale` declares constants (such as `CANADA`) that describe some common locales. This class also declares three constructors for initializing `Locale` objects in case you cannot find an appropriate `Locale` constant for a specific locale:

- `Locale(String language)` initializes a `Locale` instance to a language code, for example, "fr" for French.
- `Locale(String language, String country)` initializes a `Locale` instance to a language code and a country code, for example, "en" for English and "US" for United States.
- `Locale(String language, String country, String variant)` initializes a `Locale` instance to a language code, a country code, and a vendor- or browser-specific variant code, for example, "de" for German, "DE" for Germany, and "WIN" for Windows (or "MAC" for Macintosh).


Variant codes are useful for dealing with platform differences. For example, font differences may force you to use different characters on Windows-, Linux-, and Unix-based operating systems (such as Oracle Solaris). Unlike language and country codes, variant codes are not standardized.

Although applications can create their own `Locale` objects (perhaps to let users choose from similar locales), they will often call API methods that work with the *default locale*, which is the locale made available to the virtual machine at startup. An application can call `Locale`'s `getDefault()` class method when it needs to access this locale.

For testing or other purposes, the application can override the default locale by calling `Locale`'s `void setDefault(Locale locale)` class method. `setDefault()` sets the default locale to `locale`. However, passing `null` to `locale` causes `setDefault()` to throw `java.lang.NullPointerException`.

---

**Note**  On Android, `setDefault()` doesn’t affect the system configuration. Attempts to override the system-provided default locale may themselves be overridden by actual changes to the system configuration. Code that calls this method is usually incorrect and should be fixed by passing the appropriate locale to each locale-sensitive method that’s called.
Listing 16-17 demonstrates `getDefault()` and `setDefault()`.

**Listing 16-17. Viewing and Changing the Default Locale**

```java
import java.util.Locale;

public class MyLocale
{
    public static void main(String[] args)
    {
        System.out.println(Locale.getDefault());
        Locale.setDefault(Locale.US);
        System.out.println(Locale.getDefault());
    }
}
```

Compile Listing 16-17 as follows:

```
javac MyLocale.java
```

Now run the application as follows:

```
java MyLocale
```

When I run MyLocale, I observe the following output—my default locale is Canada (en_CA):

```
en_CA
en_US
```

You can change the default locale that's made available to the virtual machine by assigning appropriate values to the `user.language` and `user.country` system properties when you launch the application via the `java` tool. For example, the following `java` command line changes the default locale to fr_FR:

```
java -Duser.language=fr -Duser.country=FR MyLocale
```

As you continue to explore `Locale`, you'll discover additional useful methods. For example, the `String[] getISOLanguages()` class method returns an array of ISO 639 language codes (including former and changed codes) and the `String[] getISOCountries()` class method returns an array of ISO 3166 country codes.

**Resource Bundles**

An internationalized application contains no hard-coded text or other locale-specific elements (such as a specific currency format). Instead, each supported locale's version of these elements is stored outside of the application.

Java is responsible for storing each locale's version of certain elements, such as currency formats. In contrast, it's your responsibility to store each supported locale's version of other elements, such as text, audio clips, and locale-sensitive images.
Java facilitates this element storage by providing resource bundles, which are containers that hold one or more locale-specific elements, and which are each associated with one and only one locale.

Many applications work with one or more resource bundle families. Each family consists of resource bundles for all supported locales, and it typically contains one kind of element (perhaps text, or audio clips that contain language-specific verbal instructions).

Each family also shares a common family name (also known as a base name); each of its resource bundles has a unique locale designation that’s appended to the family name to differentiate one resource bundle from another within the family.

Consider an internationalized text-based game application for English and French users. After choosing game as the family name, and en and fr as the English and French locale designations, you end up with the following complete resource bundle names:

- game_en is the complete resource bundle name for English users.
- game_fr is the complete resource bundle name for French users.

Although you can store all of your game’s English text in the game_en resource bundle, you might want to differentiate between American and British text (such as elevator versus lift). This differentiation leads to the following complete resource bundle names:

- game_en_US is the complete resource bundle name for users who speak the American version of the English language.
- game_en_GB is the complete resource bundle name for users who speak the British version of the English language.

An application loads its resource bundles by calling the various getBundle() class methods that are located in the abstract java.util.ResourceBundle class. For example, the application might call the following getBundle() factory methods:

- ResourceBundle getBundle(String baseName) loads a resource bundle using the specified baseName and the default locale. For example, ResourceBundle resources = ResourceBundle.getBundle("game"); attempts to load the resource bundle whose base name is game and whose locale designation matches the default locale. When the default locale is en_US, getBundle() attempts to load game_en_US.

- ResourceBundle getBundle(String baseName, Locale locale) loads a resource bundle using the specified baseName and locale. For example, ResourceBundle resources = ResourceBundle.getBundle("game", new Locale("zh", "CN", "WIN"); attempts to load the resource bundle whose base name is game and whose locale designation is Chinese with a Windows variant. In other words, getBundle() attempts to load game_zh_CN_WIN.

**Note**  ResourceBundle is an example of a pattern that you’ll discover throughout the Internationalization APIs. With few exceptions, each API is architected around an abstract entry-point class whose class methods return instances of concrete subclasses. For this reason, these class methods are also known as factory methods.
When the resource bundle identified by the base name and locale designation doesn’t exist, the `getBundle()` methods search for the next closest bundle. For example, when the locale is en_US and `game_en_US` doesn’t exist, `getBundle()` looks for `game_en`.

The `getBundle()` methods first generate a sequence of candidate bundle names for the specified locale (language1, country1, and variant1) and the default locale (language2, country2, and variant2) in the following order:

- `baseName + "_" + language1 + "_" + country1 + "_" + variant1`
- `baseName + "_" + language1 + "_" + country1`
- `baseName + "_" + language1`
- `baseName + "_" + language2 + "_" + country2 + "_" + variant2`
- `baseName + "_" + language2 + "_" + country2`
- `baseName + "_" + language2`
- `baseName`

Candidate bundle names in which the final component is an empty string are omitted from the sequence. For example, when country1 is an empty string, the second candidate bundle name is omitted.

The `getBundle()` methods iterate over the candidate bundle names to find the first name for which they can instantiate an actual resource bundle. For each candidate bundle name, `getBundle()` attempts to create a resource bundle as follows:

- It first attempts to load a class that extends the abstract `java.util.ListResourceBundle` class using the candidate bundle name. If such a class can be found and loaded using the specified classloader, is assignment compatible with `ResourceBundle`, is accessible from `ResourceBundle`, and can be instantiated, `getBundle()` creates a new instance of this class and uses it as the result resource bundle.

- Otherwise, `getBundle()` attempts to locate a properties file. It generates a path name from the candidate bundle name by replacing all “.” characters with “/” and appending “.properties.” It attempts to find a “resource” with this name using `ClassLoader.getResource()`. (Note that a “resource” in the sense of `getResource()` has nothing to do with the contents of a resource bundle; it’s just a container of data, such as a file.) When `getResource()` finds a “resource,” it attempts to create a new `java.util.PropertyResourceBundle` instance from its contents. When successful, this instance becomes the result resource bundle.

When no result resource bundle is found, `getBundle()` throws an instance of the `java.util.MissingResourceException` class; otherwise, `getBundle()` instantiates the bundle’s parent resource bundle chain.
getBundle() builds the chain by iterating over the candidate bundle names that can be obtained by successively removing variant, country, and language (each time with the preceding “_”) from the complete resource bundle name of the result resource bundle.

With each candidate bundle name, getBundle() tries to instantiate a resource bundle as just described. When it succeeds, it calls the previously instantiated resource bundle’s setParent() method with the new resource bundle unless the previously instantiated resource bundle already has a non-null parent.

ResourceBundle declares various methods for accessing a resource bundle’s resources. For example, Object getObject(String key) gets an object for the given key from this resource bundle or one of its parent bundles.

getObject() first tries to obtain the object from this resource bundle using the protected abstract handleGetObject() method, which is implemented by concrete subclasses of ResourceBundle (such as PropertyResourceBundle).

If handleGetObject() returns null and if a non-null parent resource bundle exists, getObject() calls the parent’s getObject() method. If still not successful, it throws MissingResourceException.

Two other resource-access methods are String getString(String key) and String[] getStringArray(String key). These convenience methods are wrappers for (String) getObject(key) and (String[]) getObject(key).

**Property Resource Bundles**

A *property resource bundle* is a resource bundle backed by a *properties file*, a text file (with a .properties extension) that stores textual elements as a series of key=value entries. The key is a nonlocalized identifier that an application uses to obtain the localized value.
Note Properties files are accessed via instances of the java.util.Properties class. In Chapter 9, I mentioned that the Preferences API (discussed later in this chapter) has made Properties largely obsolete. Property resource bundles prove that the Properties class isn’t entirely obsolete.

PropertyResourceBundle, a concrete subclass of ResourceBundle, manages property resource bundles. You should rarely (if ever) need to work with this subclass. Instead, for maximum portability, you should only work with ResourceBundle, which Listing 16-18 demonstrates.

Listing 16-18. Accessing a Localized elevator Entry in game Resource Bundles

```java
import java.util.ResourceBundle;

public class PropertyResourceBundleDemo {
    public static void main(String[] args) {
        ResourceBundle resources = ResourceBundle.getBundle("game");
        System.out.println("elevator = " + resources.getString("elevator");
    }
}
```

Listing 16-18 refers to ResourceBundle instead of PropertyResourceBundle, which lets you easily migrate to ListResourceBundle as necessary. I use getString() instead of getObject() for convenience; text resources are stored in text-based properties files.

Compile Listing 16-18 as follows:

```
javac PropertyResourceBundleDemo.java
```

Run this application as follows:

```
java PropertyResourceBundleDemo
```

You’ll observe the following output:

```
Exception in thread "main" java.util.MissingResourceException: Can't find bundle for base name game, locale en_CA
    at java.util.ResourceBundle.throwMissingResourceException(Unknown Source)
    at java.util.ResourceBundle.getBundleImpl(Unknown Source)
    at java.util.ResourceBundle.getBundle(Unknown Source)
    at PropertyResourceBundleDemo.main(PropertyResourceBundleDemo.java:7)
```

This exception is thrown because no property resource bundles exist. You can easily remedy this situation by copying Listing 16-19 into a game.properties file, which is the basis for a property resource bundle.
Listing 16-19. A Fallback game.properties Resource Bundle

```java
elevator=elevator
```

Assuming that game.properties is located in the same directory as PropertyResourceBundleDemo.class, execute `java PropertyResourceBundleDemo` and you'll see the following output:

```
elevator = elevator
```

Because my locale is en_CA, `getBundle()` first tries to load `game_en_CA.properties`. Because this file doesn't exist, `getBundle()` tries to load `game_en.properties`. Because this file doesn't exist, `getBundle()` tries to load `game.properties` and succeeds.

Now copy Listing 16-20 into a file named `game_en_GB.properties`.

Listing 16-20. A game Resource Bundle for the en_GB Locale

```java
elevator=lift
```

Continue by executing the following command line:

```
java -Duser.language=en -Duser.country=GB PropertyResourceBundleDemo
```

This time, you should see the following output:

```
elevator = lift
```

With the locale set to en_GB, `getBundle()` first tries to load `game_en_GB.properties` and succeeds.

Comment out `elevator = lift` by prepending a `#` character to this line (as in `#elevator = lift`). Then execute `java -Duser.language=en -Duser.country=GB PropertyResourceBundleDemo`, and you should see the following output:

```
elevator = elevator
```

Although `getBundle()` loaded `game_en_GB.properties`, `getString()` (via `getObject()`) couldn't find an elevator entry. As a result, `getString()`/`getObject()` searched the parent resource bundle chain, encountering `game.properties`' `elevator=elevator` entry whose `elevator` value was subsequently returned.

**Note** A common reason for `getString()` throwing `MissingResourceException` in a property resource bundle context is forgetting to append `.properties` to a properties file's name.
List Resource Bundles

A list resource bundle is a resource bundle backed by a classfile, which describes a concrete subclass of ListResourceBundle (an abstract subclass of ResourceBundle). List resource bundles can store binary data (such as images or audio) as well as text. In contrast, property resource bundles can store text only.

Note    When a property resource bundle and a list resource bundle have the same complete resource bundle name, the list resource bundle takes precedence over the property resource bundle. For example, when getBundle() is confronted with game_en.properties and game_en.class, it loads game_en.class instead of game_en.properties.

Listing 16-21 demonstrates a list resource bundle by presenting a flags_en_CA class that extends ListResourceBundle.

Listing 16-21. A Resource Bundle Containing a Small Canadian Flag Image and English/French Text

```java
import java.awt.Toolkit;
import java.util.ListResourceBundle;

public class flags_en_CA extends ListResourceBundle {
    private byte image[] = {
        (byte) 137,
        (byte) 80,
        (byte) 78,
        (byte) 71,
        (byte) 13,
        (byte) 10,
        (byte) 26,
        (byte) 10,
        (byte) 0,
        (byte) 0,
        // ...
        (byte) 0,
        (byte) 0,
        (byte) 73,
        (byte) 69,
        (byte) 78,
        (byte) 68,
        (byte) 174,
        (byte) 66,
        (byte) 96,
        (byte) 130
    };
```
CHAPTER 16: Focusing on Odds and Ends

private Object[][] contents =
{
    { "flag", Toolkit.getDefaultToolkit().createImage(image) },
    { "msg", "Welcome to Canada! | Bienvenue vers le Canada!" },
    { "title", "CANADA | LA CANADA" }
};

class, which must be declared public, describes a list resource bundle whose base name is flags and whose locale designation is en_CA. This class’s image array stores a Portable Network Graphics (PNG)-based sequence of byte integers that describes an image of the Canadian flag, contents stores key/value pairs, and getContents() returns contents.

Note For brevity, Listing 16-21 doesn’t present the complete image array with Canadian flag image data. You must obtain flags_en_CA.java from this chapter’s companion code file (see the book’s introduction for instructions on how to obtain this file) to get the complete listing.

The first key/value pair consists of a key named flag (which will be passed to ResourceBundle’s getObject() method) and an instance of the java.awt.Image class. This instance represents the flag image and is obtained with the help of the java.awt.Toolkit class and its createImage() utility method.

Listing 16-22 shows you how to load the default flags_en_CA list resource bundle (or another list resource bundle via command-line arguments) and display its flag and text.

Listing 16-22. Obtaining and Displaying a List Resource Bundle’s Flag Image and Text
import java.awt.EventQueue;
import java.awt.Image;
import java.util.Locale;
import java.util.ResourceBundle;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ListResourceBundleDemo{
    public static void main(String[] args){
        Locale l = Locale.CANADA;
        if (args.length == 2)
            l = new Locale(args[0], args[1]);
final ResourceBundle resources = ResourceBundle.getBundle("flags", l);
Runnable r = new Runnable()
{
    @Override
    public void run()
    {
        Image image = (Image) resources.getObject("flag");
        String msg = resources.getString("msg");
        String title = resources.getString("title");
        ImageIcon ii = new ImageIcon(image);
        JOptionPane.showMessageDialog(null,
                                      msg,
                                      title,
                                      JOptionPane.PLAIN_MESSAGE,
                                      ii);
    }
};
EventQueue.invokeLater(r);
}

Listing 16-22’s main() method begins by selecting CANADA as its default Locale. If it detects that two arguments were passed on the command line, main() assumes that the first argument is the language code and the second argument is the country code, and it creates a new Locale object based on these arguments as its default locale.

main() next attempts to load a list resource bundle by passing the flags base name and the previously-chosen Locale object to ResourceBundle’s getBundle() method. Assuming that MissingResourceException isn’t thrown, main() creates a runnable task on which to load resources from the list resource bundle and display them graphically.

main() relies on a windowing toolkit known as Swing to present a simple user interface that displays the flag and text. Because Swing is single-threaded, where everything runs on a special thread known as the event-dispatch thread (EDT), it’s important that all Swing operations occur on this thread. EventQueue.invokeLater() makes this happen.

Note Swing is built on top of the AWT and provides many sophisticated features. This windowing toolkit was officially released as part of Java 1.2 and continues to be part of Java’s standard class library. Swing isn’t supported by Android.

Shortly after EventQueue.invokeLater() is executed on the main thread, the EDT starts running and executes the runnable. This runnable first obtains the Image object from the list resource bundle by passing flag to getObject() and casting this method’s return value to Image.

The runnable then obtains the msg and title strings by passing these keys to getString(), and it converts the Image object to a javax.swing.ImageIcon instance. This instance is required by the subsequent javax.swing.JOptionPane.showMessageDialog() method call, which presents a simple message-oriented dialog box.
Note: JOptionPane is a Swing component that makes it easy to display a standard dialog box that prompts the user to enter a value or informs the user of something important.

Now that you understand how ListResourceBundleDemo works, obtain the complete flags_en_CA.java source file and then compile its source code along with Listing 16-22, as follows:

```bash
djavac flags_en_CA.java
djavac ListResourceBundleDemo.java
```

Execute the application as follows:

```bash
djava ListResourceBundleDemo
```

Figure 16-1 shows the resulting user interface on Windows 7.

![Image of a dialog box with text in English and French]

*Figure 16-1. This almost completely localized dialog box (OK isn’t localized) displays Canada-specific resources on Windows 7*

Note: I obtained the language translations for this section’s examples from Yahoo! Babel Fish (http://babelfish.yahoo.com/), an online text translation service that no longer exists.

This book’s accompanying code file also contains flags_fr_FR.java, which presents resources localized for the France locale. After compiling this source file, execute the following command line:

```bash
djava ListResourceBundle fr FR
```

You should observe Figure 16-2 in response.
Finally, this book's accompanying code file also contains flags_ru_RU.java, which presents resources localized for the Russia locale. Compile this source file as follows:

```
javac -encoding Unicode flags_ru_RU.java
```

**Note**  Because I stored Russian characters verbatim (and not as Unicode escape sequences, such as '\u0041'), flags_ru_RU.java is a Unicode-encoded file and must be compiled with the `–encoding Unicode` option. Also, you'll need to ensure that appropriate Cyrillic fonts are installed on your platform to view the Russian text.

Execute the resulting application as follows:

```
java ListResourceBundle ru RU
```

You should see Figure 16-3 as the result.

**Taking Advantage of Cache Clearing**

Server applications are meant to run continuously; you’ll probably lose customers and get a bad reputation if these applications fail often. As a result, it’s preferable to change some aspect of their behavior interactively rather than stop and restart them.

Before Java 6, you couldn’t dynamically update the resource bundles for a server application that obtains localized text from these bundles and sends this text to clients. Because resource bundles are cached, a change to a resource bundle properties file, for example, would never be reflected in the cache and ultimately not seen by the client.
Java 6 introduced `void clearCache()` and `void clearCache(ClassLoader loader)` class methods into `ResourceBundle` that make it possible to design a server application that clears out all cached resource bundles upon command. You would clear the cache after updating the appropriate resource bundle storage, which might be a file, a database table, or some other entity that stores resource data in some format.

To demonstrate cache clearing, I've created a date-server application that sends localized text and the current date (also localized) to clients. This application's source code is shown in Listing 16-23.

**Listing 16-23. A Date Server Whose Resource Bundle Cache Can Be Cleared on Command**

```java
import java.io.Console;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.text.MessageFormat;
import java.util.Date;
import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

public class DateServer
{
    public final static int PORT = 5000;
    private ServerSocket ss;

    public DateServer(int port) throws IOException
    {
        ss = new ServerSocket(port);
    }

    public void runServer()
    {
        // This server application is console-based, as opposed to GUI-based.
        Console console = System.console();
        if (console == null)
        {
            System.err.println("unable to obtain system console");
            return;
        }
        // This would be a good place to log in the system administrator. For
        // simplicity, I've omitted this section.
        // Start a thread for handling client requests.
        Handler h = new Handler(ss);
        h.start();
    }
}
```

www.it-ebooks.info
// Receive input from system administrator; respond to exit and clear
// commands.
while (true)
{
    String cmd = console.readLine(">");
    if (cmd == null)
        continue;
    if (cmd.equals("exit"))
        System.exit(0);
    if (cmd.equals("clear"))
        h.clearRBCache();
}

public static void main(String[] args) throws IOException
{
    new DateServer(PORT).runServer();
}

class Handler extends Thread
{
    private ServerSocket ss;
    private volatile boolean doClear;

    Handler(ServerSocket ss)
    {
        this.ss = ss;
    }

    void clearRBCache()
    {
        doClear = true;
    }

    @Override
    public void run()
    {
        ResourceBundle rb = null;
        while (true)
        {
            try
            {
                // Wait for a connection.
                Socket s = ss.accept();
                // Obtain the client's locale object.
                ObjectInputStream ois;
                ois = new ObjectInputStream(s.getInputStream());
                Locale l = (Locale) ois.readObject();
                // Prepare to output message back to client.
                PrintWriter pw = new PrintWriter(s.getOutputStream());
After obtaining the console, the date server starts a handler thread to respond to clients requesting the current date formatted to their locale requirements. I discuss the `java.text.MessageFormat` class that's instantiated on this thread later in this chapter.

Following this thread's creation, you're repeatedly prompted to enter a command: clear (clear the cache) and exit (exit the application) are the only two possibilities. After changing a resource bundle, type clear to ensure that future `getBundle()` method calls initially retrieve their bundles from storage (and then the cache on subsequent method calls).

Compile Listing 16-23 as follows:

```
javac DateServer.java
```
Run this application as follows:

```java
date DateServer
```

You should observe a `>` prompt as the output.

The date server relies on resource bundles whose base name is `datetemplate`. I’ve created two bundles, which are stored in files named `datemsg_en.properties` and `datemsg_fr.properties`. The contents of the former file appear in Listing 16-24.

**Listing 16-24. The contents of `datemsg_en.properties`**

```
datetemplate = The date is {0, date, long}.
```

After connecting to the date server, a date-client application sends the server a `Locale` object; the client receives a `String` object in response. When the date server doesn’t support the locale (a resource bundle cannot be found), it returns a string consisting of a single question mark. Otherwise, the date server returns a string consisting of localized text. Listing 16-25 presents the source code to a simple date-client application.

**Listing 16-25. Communicating with the Date Server**

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Locale;

public class DateClient
{
    final static int PORT = 5000;

    public static void main(String[] args)
    {
        try
        {
            // Establish a connection to the date server. For simplicity, the
            // server is assumed to run on the same machine as the client. The
            // PORT constants of both server and client must be the same.
            Socket s = new Socket("localhost", PORT);
            // Send the default locale to the date server.
            ObjectOutputStream oos;
            oos = new ObjectOutputStream(s.getOutputStream());
            oos.writeObject(Locale.getDefault());
            // Obtain and output the server's response.
            InputStreamReader isr;
            isr = new InputStreamReader(s.getInputStream());
            BufferedReader br = new BufferedReader(isr);
            System.out.println(br.readLine());
        }
    }
}
```
catch (Exception e) {
    System.err.println(e);
}
}

Compile Listing 16-25 as follows:

javac DateClient.java

Assuming that you've previously started the date server, execute the following command to run DateClient:

java DateClient

You should observe the current date in the default locale in reply.

For simplicity, the date client sends the default locale to the server. You can override this locale via the `java` tool’s `-D` command-line option. For example, execute the following command to send a `Locale("fr", ")` object to the server:

java -Duser.language=fr -Duser.country="" DateClient

You should receive a reply in French, as demonstrated here:

La date est 21 décembre 2013.

You can verify the usefulness of cache clearing by performing a simple experiment with the date server and date client applications. Before you begin this experiment, create a second copy of Listing 16-24 in which “Thee” replaces “The”. Make sure that the properties file containing Thee is in the same directory as the date server, and then follow these steps:

1. Start the date server.
2. Run the client using en as the locale (via `java DateClient` when English is the default locale or via `java -Duser.language=en DateClient` otherwise). You should see a message beginning with “Thee date is.”
3. Copy the Listing 16-24 properties file to the server’s directory.
4. Type `clear` at the server prompt.
5. Run the client using en as the locale. This time, you should see a message beginning with “The date is.”
Taking Control of the `getBundle()` Methods

Before Java 6, `ResourceBundle`'s `getBundle()` methods were hardwired to look for resource bundles as follows:

- **Look for certain kinds of bundles**: Properties-based or class-based.
- **Look in certain places**: Properties files or classfiles whose directory paths are indicated by fully qualified resource bundle base names.
- **Use a specific search strategy**: When a search based on a specified locale fails, perform the search using the default locale.
- **Use a specific loading procedure**: When a class and a properties file share the same candidate bundle name, the class is always loaded while the properties file remains hidden.

Furthermore, resource bundles were always cached.

Because this lack of flexibility prevents you from performing tasks such as obtaining resource data from sources other than properties files and classfiles (such as an XML file or a database), Java 6 reworked `ResourceBundle` to depend on a nested `Control` class. This nested class provides several callback methods that are invoked during the resource bundle search-and-load process. By overriding specific callback methods, you can achieve the desired flexibility. When none of these methods are overridden, the `getBundle()` methods behave as they always have.

Control offers the following methods:

- `List<Locale> getCandidateLocales(String baseName, Locale locale)` returns a list of candidate locales for the specified `baseName` and `locale`. `NullPointerException` is thrown when `baseName` or `locale` is null.
- `static ResourceBundle.Control getControl(List<String> formats)` returns a `ResourceBundle.Control` instance whose `getFormats()` method returns the specified formats. `NullPointerException` is thrown when the `formats` list is null and `java.lang.IllegalArgumentException` is thrown when the list of formats isn't known.
- `Locale getFallbackLocale(String baseName, Locale locale)` returns a fallback locale for further resource bundle searches (via `ResourceBundle.getBundle()`). `NullPointerException` is thrown when `baseName` or `locale` is null.
- `List<String> getFormats(String baseName)` returns a list of strings that identify the formats to be used in loading resource bundles that share the given `baseName`. `NullPointerException` is thrown when `baseName` is null.

**Caution** It's tempting to always want to invoke `clearCache()` before invoking `getBundle()`. However, doing so negates the performance benefit that caching brings to an application. For this reason, you should use `clearCache()` sparingly as the date server application demonstrates.
static final ResourceBundle.Control getNoFallbackControl(List<String> formats) returns a ResourceBundle.Control instance whose getFormats() method returns the specified formats and whose getFallBackLocale() method returns null. NullPointerException is thrown when the formats list is null and IllegalArgumentException is thrown when the list of formats isn’t known.

long getTimeToLive(String baseName, Locale locale) returns the time-to-live value for resource bundles loaded via this ResourceBundle.Control instance. NullPointerException is thrown when baseName or locale is null.

boolean needsReload(String baseName, Locale locale, String format, ClassLoader loader, ResourceBundle bundle, long loadTime) determines when the expired cached bundle needs to be reloaded by comparing the last modified time with loadTime. It returns a true value (the bundle needs to be reloaded) when the last modified time is more recent then the loadTime. NullPointerException is thrown when baseName, locale, format, loader, or bundle is null.

ResourceBundle newBundle(String baseName, Locale locale, String format, ClassLoader loader, boolean reload) creates a new resource bundle based on a combination of baseName and locale, and takes the format and loader into consideration. NullPointerException is thrown when baseName, locale, format, or loader is null (or when toBundleName(), which is called by this method, returns null). IllegalArgumentException is thrown when format is unknown or when the resource identified by the given parameters contains malformed data; java.lang.ClassCastException is thrown when the loaded class cannot be cast to ResourceBundle; java.lang.IllegalAccessException is thrown when the class or its noargument constructor isn’t accessible; java.langInstantiationException is thrown when the class cannot be instantiated for some other reason; java.lang.ExceptionInInitializerError is thrown when the class’s static initializer fails; and IOException is thrown when an I/O error occurs while reading resources using any I/O operations.

String toBundleName(String baseName, Locale locale) converts the specified baseName and locale into a bundle name whose components are separated by underscore characters. For example, when baseName is MyResources and locale is en, the resulting bundle name is MyResources_en. NullPointerException is thrown when baseName or locale is null.

String toResourceName(String bundleName, String suffix) converts the specified bundleName to a resource name. Forward-slash separators replace package period separators; a period followed by suffix is appended to the resulting name. For example, when bundleName is com.company.MyResources_en and suffix is properties, the resulting resource name is com/company/MyResources_en.properties. NullPointerException is thrown when bundleName or suffix is null.

The getCandidateLocales() method is called by a ResourceBundle.getBundle() class method each time the class method looks for a resource bundle for a target locale. You can override getCandidateLocales() to modify the target locale’s parent chain. For example, when you want your Hong Kong resource bundles to share traditional Chinese strings, make Chinese/Taiwan
resource bundles the parent bundles of Chinese/Hong Kong resource bundles. *The Java Tutorial’s “Customizing Resource Bundle Loading” lesson* (http://download.oracle.com/javase/tutorial/i18n/resbundle/control.html) shows how to accomplish this task.

The `getFallbackLocale()` method is called by a `ResourceBundle.getBundle()` class method each time the class method cannot find a resource bundle based on `getFallbackLocale()`’s `baseName` and `locale` arguments. You can override this method to return null when you don’t want to continue a search using the default locale.

The `getFormats()` method is called by a `ResourceBundle.getBundle()` class method when it needs to load a resource bundle that’s not found in the cache. This returned list of formats determines if the resource bundles being sought during the search are classfiles only, properties files only, both classfiles and properties files, or some other application-defined formats. When you override `getFormats()` to return application-defined formats, you’ll also need to override `newBundle()` to load bundles based on these formats.

Earlier, I demonstrated using `clearCache()` to remove all resource bundles from `ResourceBundle`’s cache. Rather than explicitly clearing the cache, you can control how long resource bundles remain in the cache before they need to be reloaded by using the `getTimeToLive()` and `needsReload()` methods. The `getTimeToLive()` method returns one of the following values:

- A positive value representing the number of milliseconds that resource bundles loaded under the current `ResourceBundle.Control` instance can remain in the cache without being validated against their source data.
- 0 when the bundles must be validated each time they are retrieved from the cache.
- `ResourceBundle.Control.TTL_DONT_CACHE` when the bundles are not cached.
- `ResourceBundle.Control.TTL_NO_EXPIRATION_CONTROL` when the bundles are not to be removed from the cache under any circumstance (apart from low memory or when you explicitly clear the cache).

When a `ResourceBundle.getBundle()` class method finds an expired resource bundle in the cache, it calls `needsReload()` to determine whether or not the resource bundle should be reloaded. When this method returns true, `getBundle()` removes the expired resource bundle from the cache; a false return value updates the cached resource bundle with the time-to-live value returned from `getTimeToLive()`.

The `toBundleName()` method is called from the default implementations of `needsReload()` and `newBundle()` when they need to convert a base name and a locale to a bundle name. You can override this method to load resource bundles from different packages instead of the same package.

For example, assume that `MyResources.properties` stores your application’s default (base) resource bundle and that you also have a `MyResources_de.properties` file for storing your application’s German language resources. The default implementation of `ResourceBundle.Control` organizes these bundles in the same package. By overriding `toBundleName()` to change how these bundles are named, you can place them into different packages.

For example, you could have a `com.company.app.i18n.base.MyResources` package corresponding to the `com/company/app/i18n/base/MyResources.properties` resource file and a `com.company.app.i18n.de.MyResources` package corresponding to the `com/company/app/i18n/de/MyResources.properties` file.
CHAPTER 16: Focusing on Odds and Ends

You can learn how to do this by exploring a similar example in the Oracle/Sun Developer Network “International Enhancements in Java SE 6” article (www.oracle.com/technetwork/articles/javase/i18n-enhance-137163.html).

Although you will often subclass ResourceBundle.Control and override some combination of the callback methods, this isn’t always necessary. For example, when you want to restrict resource bundles to classfiles only or to properties files only, you can invoke getControl() to return a ready-made ResourceBundle.Control (thread-safe singleton) object that takes care of this task. To get this object, you will need to pass one of the following ResourceBundle.Control constants to getControl():

- **FORMAT_PROPERTIES**, which describes an unmodifiable java.util.List<String> containing "java.properties".
- **FORMAT_CLASS**, which describes an unmodifiable List<String> containing "java.class".
- **FORMAT_DEFAULT**, which describes an unmodifiable List<String> containing "java.class" followed by "java.properties".

The first example in ResourceBundle.Control’s JDK documentation uses getControl() to return a ResourceBundle.Control instance that restricts resource bundles to properties files.

You can also invoke getNoFallbackControl() to return a ready-made ResourceBundle.Control instance that (in addition to restricting resource bundles to classfiles or properties files only) tells the new getBundle() methods to avoid falling back to the default locale when searching for a resource bundle. The getNoFallbackControl() method recognizes the same formats argument as getControl(); it returns a thread-safe singleton whose getFallbackLocale() method returns null.

### Break Iterators

Internationalized text-processing applications (such as word processors) need to detect logical boundaries within the text they’re manipulating. For example, a word processor needs to detect these boundaries when highlighting a character, selecting a word to cut to the clipboard, moving the caret (text insertion point indicator) to the start of the next sentence, and wrapping a word at the end of a line.

Java provides the Break Iterator API with its abstract java.text.BreakIterator entry-point class to detect text boundaries.

BreakIterator declares the following class methods for obtaining break iterators that detect character, word, sentence, and line boundaries:

- BreakIterator getCharacterInstance()
- BreakIterator getWordInstance()
- BreakIterator getSentenceInstance()
- BreakIterator getLineInstance()

Each of these class methods returns a break iterator for the default locale.
When you need a break iterator for a specific locale, you can call the following class methods:

- `BreakIterator getCharacterInstance(Locale locale)`
- `BreakIterator getWordInstance(Locale locale)`
- `BreakIterator getSentenceInstance(Locale locale)`
- `BreakIterator getLineInstance(Locale locale)`

Each of these class methods throws `NullPointerException` when its locale argument is `null`.

`BreakIterator`'s locale-sensitive class methods might not support every locale. For this reason, you should only pass `Locale` objects that are also stored in the array returned from this class's `Locale[] getAvailableLocales()` class method (which is also declared in other entry-point classes) to the aforementioned class methods; this array contains at least `Locale.US`. Check out the following example:

```java
Locale[] supportedLocales = BreakIterator.getAvailableLocales();
BreakIterator bi = BreakIterator.getCharacterInstance(supportedLocales[0]);
```

This example obtains `BreakIterator`'s supported locales and passes the first locale (possibly `Locale.US`) to `getCharacterInstance(Locale)`.

A `BreakIterator` instance has an imaginary cursor that points to the current boundary within a text string. This cursor position can be interrogated and the cursor moved from boundary to boundary with the help of the following `BreakIterator` methods:

- `int current()` returns the text boundary that was most recently returned by `next()`, `next(int)`, `previous()`, `first()`, `last()`, `following(int)`, or `preceding(int)`. When any of these methods returns `BreakIterator.DONE` because either the first or the last text boundary has been reached, `current()` returns the first or last text boundary depending on which one was reached.

- `int first()` returns the first text boundary. The iterator's current position is set to this boundary.

- `int following(int offset)` returns the first text boundary following the specified character offset. When `offset` equals the last text boundary, `following(int)` returns `BreakIterator.DONE` and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the returned text boundary. The value returned is always greater than `offset` or `BreakIterator.DONE`.

- `int last()` returns the last text boundary. The iterator's current position is set to this boundary.

- `int next()` returns the text boundary following the current boundary. When the current boundary is the last text boundary, `next()` returns `BreakIterator.DONE` and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the boundary following the current boundary.
int next(int n) returns the nth text boundary from the current boundary. When either the first or the last text boundary has been reached, next(int) returns BreakIterator.DONE and the current position is set either to the first or last text boundary depending on which one is reached. Otherwise, the iterator's current position is set to the new text boundary.

int preceding(int offset) returns the last text boundary preceding the specified character offset. When offset equals the first text boundary, preceding(int) returns BreakIterator.DONE and the iterator's current position is unchanged. Otherwise, the iterator's current position is set to the returned text boundary. The returned value is always less than offset or equals BreakIterator.DONE. (This method was added to BreakIterator in Java 1.2. It couldn't be declared abstract because abstract methods cannot be added to existing classes; such methods would also have to be implemented in subclasses that might be inaccessible.)

int previous() returns the text boundary preceding the current boundary. When the current boundary is the first text boundary, previous() returns BreakIterator.DONE and the iterator’s current position is unchanged. Otherwise, the iterator’s current position is set to the boundary preceding the current boundary.

Figure 16-4 reveals that characters are located between boundaries, boundaries are zero-based, and the last boundary is the length of the string.

Listing 16-26 shows how to use a character-based break iterator to iterate over a string’s characters in a locale-independent manner.

Listing 16-26. Iterating Over English/US and Arabic/Saudi Arabia Strings

```java
import java.text.BreakIterator;
import java.util.Locale;

public class BreakIteratorDemo
{
    public static void main(String[] args)
    {
        BreakIterator bi = BreakIterator.getCharacterInstance(Locale.US);
        bi.setText("JAVA");
        dumpPositions(bi);
    }
}
```
Listing 16-26’s `main()` method first obtains a character-based break iterator for the United States locale. `main()` then calls the iterator’s `setText()` method to specify `JAVA` as the text to be iterated over.

Iteration occurs in the `dumpPositions()` method. After calling `first()` to obtain the first boundary, this method uses a `while` loop to output the boundary and move to the next boundary (via `next()`) while the current boundary doesn’t equal `BreakIterator.DONE`.

Because character iteration is straightforward for English words, `main()` next obtains a character-based break iterator for the Saudi Arabia locale and uses this iterator to iterate over the characters in Figure 16-5’s Arabic version of “shelf” (as in shelf of books).

![Figure 16-5](image)

In Arabic, the word “shelf” consists of letters resh and pe, and diacritic shadda. A *diacritic* is an ancillary *glyph*, or mark on paper or other writing medium, added to a letter, or basic glyph. Shadda, which is shaped like a small written Latin w, indicates *gemination* (consonant doubling or extra length), which is *phonemic* (the smallest identifiable discrete unit of sound employed to form meaningful contrasts between utterances) in Arabic. Shadda is written above the consonant that’s to be doubled, which happens to be pe in this example.

Compile Listing 16-26 as follows:

```bash
javac BreakIteratorDemo.java
```
Run this application as follows:

```java
java BreakIteratorDemo
```

You observe the following output:

```
0 1 2 3 4
0 1 3
```

The first output line reveals Figure 16-4’s character boundaries for the word JAVA. The second output line (0 comes before resh, 1 comes before pe) implies that you cannot move an Arabic word processor’s caret on the screen once for every Unicode character. Instead, it’s moved once for every user character, a logical character that can be composed of multiple Unicode characters, such as pe (\u0641) and shadda (\u0651).

---

**Note**  For examples of break iterators that iterate over words, sentences, and lines, check out the “Detecting Text Boundaries” section ([http://download.oracle.com/javase/tutorial/i18n/text/boundaryintro.html](http://download.oracle.com/javase/tutorial/i18n/text/boundaryintro.html)) in *The Java Tutorials*.

---

**Collators**

Applications perform string comparisons while sorting text. When an application targets English-oriented users, `String`’s `compareTo()` method is probably sufficient for comparing strings. However, this method’s binary comparison of each string’s Unicode characters isn’t reliable for languages where the relative order of their characters doesn’t correspond to the Unicode values of these characters. French is one example.

Java provides the Collator API with its abstract `java.text.Collator` entry-point class for making reliable comparisons. `Collator` declares the following class methods for obtaining collators:

- `Collator getInstance()`
- `Collator getInstance(Locale locale)`

The first class method obtains a collator for the default locale; the second class method throws `NullPointerException` when its `locale` argument is null. As with BreakIterator, you should only pass `Locale` objects that are also stored in the array returned from `Collator`'s `Locale[] getAvailableLocales()` class method to `getInstance(Locale)`.

Listing 16-27 shows how to use a collator to perform comparisons so that French words differing only in terms of accented characters are sorted into the correct order.
Listing 16-27. Using a Collator to Order French Words Correctly in the France Locale

```java
import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;

public class CollatorDemo
{
    public static void main(String[] args)
    {
        Collator en_USCollator = Collator.getInstance(Locale.US);
        Collator fr_FRCollator = Collator.getInstance(Locale.FRANCE);
        String[] words =
        {
            "côte", "coté", "côté", "cote"
        };
        Arrays.sort(words, en_USCollator);
        for (String word: words)
        {
            System.out.println(word);
        }
        System.out.println();
        Arrays.sort(words, fr_FRCollator);
        for (String word: words)
        {
            System.out.println(word);
        }
    }
}
```

In Listing 16-27, each of the four words being sorted has a different meaning. For example, côte means coast and côté means side.

Compile Listing 16-27 as follows:

```
javac CollatorDemo.java
```

Run this application as follows:

```
java CollatorDemo
```

I observe the following output in Windows Notepad:

```
cote
coté
côte
côté
cote
coté
côté
côté
```
The first four output lines show the order in which these words are sorted according to the en_US locale. This ordering isn’t correct because it doesn’t account for accents. In contrast, the final four output lines show the correct order when the words are sorted according to the fr_FR locale. Words are compared as if none of the characters contain accents and then equal words are compared from right to left for accents.

**Note** Learn about Collator’s java.text.RuleBasedCollator subclass for creating custom collators when predefined collation rules don’t meet your needs, and about improving collation performance via java.text.CollationKey and Collator’s CollationKey getCollationKey(String source) method, by reading the “Comparing Strings” section (http://download.oracle.com/javase/tutorial/i18n/text/collationintro.html) in The Java Tutorials.

**Dates, Time Zones, and Calendars**

Internationalized applications must properly handle dates, time zones, and calendars. A date is a recorded temporal moment, a time zone is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a calendar is a system of organizing the passage of time.

**Note** GMT identifies the standard geographical location from where all time is measured. UTC, which stands for Coordinated Universal Time, is often specified in place of GMT.

Java 1.0 introduced the java.util.Date class as its first attempt to describe calendars. However, Date was not amenable to internationalization because of its English-oriented nature and because of its inability to represent dates prior to midnight January 1, 1970 GMT, which is known as the Unix epoch (the date when Unix began to be used).

Date was eventually refactored to make it more useful by allowing Date instances to represent dates before the epoch as well as after the epoch, and by deprecating most of this class’s constructors and methods; deprecated methods have been replaced by more appropriate API classes. Table 16-2 describes the more useful Date class.
Table 16-2. Date Constructors and Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date()</td>
<td>Allocates a Date object and initializes it to the current time by calling System.currentTimeMillis().</td>
</tr>
<tr>
<td>Date(long date)</td>
<td>Allocates a Date object and initializes it to the time represented by date milliseconds. A negative value indicates a time before the epoch, 0 indicates the epoch, and a positive value indicates a time after the epoch.</td>
</tr>
<tr>
<td>boolean after(Date date)</td>
<td>Returns true when this date occurs after date. This method throws NullPointerException when date is null.</td>
</tr>
<tr>
<td>boolean before(Date date)</td>
<td>Returns true when this date occurs before date. This method throws NullPointerException when date is null.</td>
</tr>
<tr>
<td>Object clone()</td>
<td>Returns a copy of this object.</td>
</tr>
<tr>
<td>int compareTo(Date date)</td>
<td>Compares this date with date. Returns 0 when this date equals date, a negative value when this date comes before date, and a positive value when this date comes after date. This method throws NullPointerException when date is null.</td>
</tr>
<tr>
<td>boolean equals(Object obj)</td>
<td>Compares this date with the Date object represented by obj. Returns true if and only if obj isn't null and is a Date object that represents the same point in time (to the millisecond) as this date.</td>
</tr>
<tr>
<td>long getTime()</td>
<td>Returns the number of milliseconds that must elapse before the epoch (a negative value) or have elapsed since the epoch (a positive value).</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns this date's hash code. The result is the exclusive OR of the two halves of the long integer value returned by getTime(). That is, the hash code is the value of expression (int) (this.getTime() ^ (this.getTime() &gt;&gt;&gt; 32)).</td>
</tr>
<tr>
<td>void setTime(long time)</td>
<td>Sets this date to represent the point in time specified by time milliseconds (a negative value refers to before the epoch; a positive value refers to after the epoch).</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a String object containing this date’s representation as dow mon dd hh:mm:ss zzz yyyy, where dow is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat), mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec), dd is the two decimal-digit day of the month (01 through 31), hh is the two decimal-digit hour of the day (00 through 23), mm is the two decimal-digit minute within the hour (00 through 59), ss is the two decimal-digit second within the minute (00 through 61, where 60 and 61 account for leap seconds), zzz is the (possibly empty) time zone (and may reflect daylight saving time), and yyyy is the four decimal-digit year.</td>
</tr>
</tbody>
</table>
Listing 16-28 provides a small demonstration of the Date class.

Listing 16-28. Exploring the Date Class

```java
import java.util.Date;

public class DateDemo
{
    public static void main(String[] args)
    {
        Date now = new Date();
        System.out.println(now);
        Date later = new Date(now.getTime() + 86400);
        System.out.println(later);
        System.out.println(now.after(later));
        System.out.println(now.before(later));
    }
}
```

Listing 16-28’s `main()` method creates a pair of `Date` objects (`now` and `later`) and outputs their dates, formatted according to `Date`’s implicitly called `toString()` method. `main()` then demonstrates `after()` and `before()`, proving that `now` comes before `later`, which is one second in the future.

Compile Listing 16-28 as follows:

`javac DateDemo.java`

Run this application as follows:

`java DateDemo`

You should observe output similar to the following:

```
Sat Jan 04 13:25:20 CST 2014
Sat Jan 04 13:26:47 CST 2014
false
true
```

Date’s `toString()` method reveals that a time zone is part of a date. Java provides the abstract `java.util.TimeZone` entry-point class for obtaining instances of `TimeZone` subclasses. This class declares a pair of class methods for obtaining these instances:

- `TimeZone getDefault()`
- `TimeZone getTimeZone(String ID)`

The latter method returns a `TimeZone` instance for the time zone whose `String` identifier (such as "CST") is passed to `ID`. 
Note Some time zones take into account *daylight saving time*, the practice of temporarily advancing clocks so that afternoons have more daylight and mornings have less, for example, Central Daylight Time (CDT). Check out Wikipedia’s “Daylight saving time” entry (http://en.wikipedia.org/wiki/Daylight_saving_time) to learn more about daylight saving time.

When you need to introduce a new time zone or modify an existing time zone, perhaps to deal with changes to a time zone’s daylight saving time policy, you can work directly with TimeZone’s java.util. SimpleTimeZone concrete subclass. SimpleTimeZone describes a raw offset from GMT and provides rules for specifying the start and end of daylight saving time.

Java 1.1 introduced the Calendar API with its abstract java.util.Calendar entry-point class as a replacement for Date. Calendar is intended to represent any kind of calendar. However, time constraints meant that only the Gregorian calendar could be implemented (via the concrete java.util.GregorianCalendar subclass) for version 1.1.

Note Java 1.4 introduced support for the Thai Buddhist calendar via an internal class that subclasses Calendar. Java 6 introduced support for the Japanese Imperial Era calendar via the package-private java.util.JapaneseImperialCalendar class, which also subclasses Calendar.

Calendar declares the following class methods for obtaining calendars:

- Calendar getInstance()
- Calendar getInstance(Locale locale)
- Calendar getInstance(TimeZone zone)
- Calendar getInstance(TimeZone zone, Locale locale)

The first and third methods return calendars for the default locale; the second and fourth methods take the specified locale into account. Also, calendars returned by the first two methods are based on the current time in the default time zone; calendars returned by the last two methods are based on the current time in the specified time zone.

Calendar declares various constants, including YEAR, MONTH, DAY_OF_MONTH, DAY_OF_WEEK, LONG, and SHORT. These constants identify the year (four digits), month (0 represents January), current month day (1 through the month’s last day), and current weekday (1 represents Sunday) calendar fields, and display styles (such as January versus Jan).

The first four constants are used with Calendar’s various set() methods to set calendar fields to specific values (set the year field to 2012, for example). They’re also used with Calendar’s int get(int field) method to return field values, along with other field-oriented methods such as void clear(int field) (unset a field).
The latter two constants are used with Calendar's String getDisplayName(int field, int style, Locale locale) and Map<String,Integer> getDisplayNames(int field, int style, Locale locale) methods, which return short (Jan, for example) or long (January, for example) localized String representations of various field values.

Listing 16-29 shows how to use various Calendar constants and methods to output calendar pages according to the en_US and fr_FR locales.

Listing 16-29. Outputting Calendar Pages

```java
import java.util.Calendar;
import java.util.Iterator;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

public class CalendarDemo {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.err.println("usage: java CalendarDemo yyyy mm [f|F]");
            return;
        }
        try {
            int year = Integer.parseInt(args[0]);
            int month = Integer.parseInt(args[1]);
            Locale locale = Locale.US;
            if (args.length == 3 && args[2].equalsIgnoreCase("f"))
                locale = Locale.FRANCE;
            showPage(year, month, locale);
        }
        catch (NumberFormatException nfe) {
            System.err.print(nfe);
        }
    }

    static void showPage(int year, int month, Locale locale) {
        if (month < 1 || month > 12)
            throw new IllegalArgumentException("month [" + month + "] out of " + "range [1, 12]");
        Calendar cal = Calendar.getInstance(locale);
        cal.set(Calendar.YEAR, year);
        cal.set(Calendar.MONTH, --month);
        cal.set(Calendar.DAY_OF_MONTH, 1);
        displayMonthAndYear(cal, locale);
        displayWeekdayNames(cal, locale);
    }
}
```
```java
int daysInMonth = cal.getActualMaximum(Calendar.DAY_OF_MONTH);
int firstRowGap = cal.get(Calendar.DAY_OF_WEEK) - 1; // 0 = Sunday
for (int i = 0; i < firstRowGap; i++)
    System.out.print("   ");
for (int i = 1; i <= daysInMonth; i++)
{
    if (i < 10)
        System.out.print(' ');
    System.out.print(i);
    if ((firstRowGap + i) % 7 == 0)
        System.out.println();
    else
        System.out.print(' ');
}
System.out.println();
}

static void displayMonthAndYear(Calendar cal, Locale locale)
{
    System.out.println(cal.getDisplayName(Calendar.MONTH, Calendar.LONG, locale) + " " + cal.get(Calendar.YEAR));
}

static void displayWeekdayNames(Calendar cal, Locale locale)
{
    Map<String, Integer> weekdayNamesMap;
    weekdayNamesMap = cal.getDisplayNames(Calendar.DAY_OF_WEEK, Calendar.SHORT, locale);
    String[] names = new String[weekdayNamesMap.size()];
    int[] indexes = new int[weekdayNamesMap.size()];
    Set<Map.Entry<String, Integer>> weekdayNamesEntries;
    weekdayNamesEntries = weekdayNamesMap.entrySet();
    Iterator<Map.Entry<String, Integer>> iter;
    iter = weekdayNamesEntries.iterator();
    while (iter.hasNext())
    {
        Map.Entry<String, Integer> entry = iter.next();
        names[entry.getValue() - 1] = entry.getKey();
        indexes[entry.getValue() - 1] = entry.getValue();
    }
    for (int i = 0; i < names.length; i++)
        for (int j = i; j < names.length; j++)
            if (indexes[j] == i + 1)
            {
                System.out.print(names[j].substring(0, 2) + " ");
                continue;
            }
    System.out.println();
}
```
Listing 16-29 is pretty straightforward with the exception of displayWeekdayNames(). This method calls Calendar's getDisplayNames() method to return a map of localized weekday names. Instead of returning a map where the keys are java.lang.Integers and the values are localized Strings, this map's keys are the localized Strings.

This would be fine if the keys were ordered (as in Sunday first and Saturday last, or lundi first and dimanche last). However, they're not ordered. To output these names in order, it's necessary to obtain a set of map entries, iterate over these entries and populate parallel arrays, and then iterate over these arrays to output the weekday names.

Note A French calendar begins the week on lundi (Monday) and ends it on dimanche (Sunday). However, Calendar doesn't take this ordering into account.

Compile Listing 16-29 as follows:

javac CalendarDemo.java

Run this application with an appropriate year, month, and locale:

java CalendarDemo 2014 01

When I specify the previous command line for the en_CA locale, I observe the following calendar page:

<table>
<thead>
<tr>
<th>January 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>Su</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>29</td>
</tr>
</tbody>
</table>

If you would like to see this page in the fr_FR locale, specify either of the following command lines:

java CalendarDemo 2014 01 f
java CalendarDemo 2014 01 F

You should then observe the following calendar page:

<table>
<thead>
<tr>
<th>janvier 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>di</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>29</td>
</tr>
</tbody>
</table>
Calendar declares a `Date getTime()` method that returns a calendar's time representation as a `Date` instance. Calendar also declares a `void setTime(Date date)` method that sets a calendar's time representation to the specified date.

Formatters

Internationalized applications don't present unformatted numbers (including currencies and percentages), dates, and messages to the user. These items must be formatted according to the user's locale so that they appear meaningful. To help with formatting, Java provides the abstract `java.text.Format` class and various subclasses.

Number Formatters

The abstract `java.text.NumberFormat` entry-point class (a `Format` subclass) declares the following class methods to return formatters that format numbers as currencies, integers, numbers with decimal points, and percentages (and also to parse such values):

- `NumberFormat getCurrencyInstance()`
- `NumberFormat getCurrencyInstance(Locale locale)`
- `NumberFormat getIntegerInstance()`
- `NumberFormat getIntegerInstance(Locale locale)`
- `NumberFormat getInstance()`
- `NumberFormat getInstance(Locale locale)`
- `NumberFormat getNumberInstance()`
- `NumberFormat getNumberInstance(Locale locale)`
- `NumberFormat getPercentInstance()`
- `NumberFormat getPercentInstance(Locale locale)`

The `getInstance()` and `getInstance(Locale)` class methods are equivalent to `getNumberInstance()` and `getNumberInstance(Locale)`. They're present as a shorthand convenience to the longer-named `getNumberInstance()` methods.

Listing 16-30 shows you how to obtain and use number formatters to format numbers as currencies, integers, numbers with decimal points, and percentages for various locales.
Listing 16-30. Formatting Numbers as Currencies, Integers, Numbers with Decimal Points, and Percentages

```java
import java.text.NumberFormat;
import java.util.Locale;

global class NumberFormatDemo
{
    public static void main(String[] args)
    {
        System.out.println("Unformatted: " + 9875432.25);
        formatCurrencies(Locale.US, 9875432.25);
        formatCurrencies(Locale.FRANCE, 9875432.25);
        formatCurrencies(Locale.GERMANY, 9875432.25);
        System.out.println();
        System.out.println("Unformatted: " + 123456789.0);
        formatIntegers(Locale.US, 123456789.0);
        formatIntegers(Locale.FRANCE, 123456789.0);
        formatIntegers(Locale.GERMANY, 123456789.0);
        System.out.println();
        System.out.println("Unformatted: " + 6751.326);
        formatNumbers(Locale.US, 6751.326);
        formatNumbers(Locale.FRANCE, 6751.326);
        formatNumbers(Locale.GERMANY, 6751.326);
        System.out.println();
        System.out.println("Unformatted: " + 0.85);
        formatPercentages(Locale.US, 0.85);
        formatPercentages(Locale.FRANCE, 0.85);
        formatPercentages(Locale.GERMANY, 0.85);
    }

    static void formatCurrencies(Locale locale, double amount)
    {
        NumberFormat nf = NumberFormat.getCurrencyInstance(locale);
        System.out.println(locale + " : " + nf.format(amount));
    }

    static void formatIntegers(Locale locale, double amount)
    {
        NumberFormat nf = NumberFormat.getIntegerInstance(locale);
        System.out.println(locale + " : " + nf.format(amount));
    }

    static void formatNumbers(Locale locale, double amount)
    {
        NumberFormat nf = NumberFormat.getNumberInstance(locale);
        System.out.println(locale + " : " + nf.format(amount));
    }
    ```
static void formatPercentages(Locale locale, double amount)
{
    NumberFormat nf = NumberFormat.getPercentInstance(locale);
    System.out.println(locale + " : " + nf.format(amount));
}

Listing 16-30 uses a double instead of a java.math.BigDecimal object to represent 9875432.25 as a currency, for simplicity and because this value can be represented exactly as a double.

Compile Listing 16-30 as follows:

javac NumberFormatDemo.java

Run this application as follows:

java NumberFormatDemo

Figure 16-6 shows the resulting output in the Windows Notepad editor.

Figure 16-6. Windows Notepad reveals unformatted and formatted numeric output for the US, France, and Germany locales

NumberFormat declares void setMaximumFractionDigits(int newValue), void setMaximumIntegerDigits(int newValue), void setMinimumFractionDigits(int newValue), and void setMinimumIntegerDigits(int newValue) methods to limit the number of digits that are
allowed in a formatted number's integer or fraction. These methods are helpful for aligning numbers, as the following example demonstrates:

NumberFormat nf = NumberFormat.getInstance();
System.out.println(nf.format(123.4567)); // I observe 123.457
nf.setMaximumIntegerDigits(10);
nf.setMinimumIntegerDigits(6);
nf.setMaximumFractionDigits(2);
nf.setMinimumFractionDigits(2);
System.out.println(nf.format(123.4567)); // I observe 000,123.46
System.out.println(nf.format(80978.3));  // I observe 080,978.30

This example specifies that a number’s integer portion cannot exceed ten digits but must have a minimum of six digits. Leading zeros are output to meet the minimum. The example reveals that the fraction is rounded.

A concrete subclass of NumberFormat might enforce an upper limit on the value passed to setMaximumFractionDigits(int), setMaximumIntegerDigits(int), setMinimumFractionDigits(int), or setMinimumIntegerDigits(int). Call getMaximumFractionDigits(), getMaximumIntegerDigits(), getMinimumFractionDigits(), or getMinimumIntegerDigits() to find out if the value you specified has been accepted.

Note When you need to create customized number formatters, you’ll find yourself working with NumberFormat’s java.text.DecimalFormat subclass and this subclass’s java.text.DecimalFormatSymbols companion class. The “Customizing Formats” section (http://download.oracle.com/javase/tutorial/i18n/format/decimalFormat.html) in The Java Tutorials introduces you to these classes.

Date Formatters

The abstract java.text.DateFormat entry-point class (a Format subclass) provides access to formatters that format Date instances as dates or time values (and also to parse such values). This class declares the following class methods:

- DateFormat getDateInstance()
- DateFormat getDateInstance(int style)
- DateFormat getDateInstance(int style, Locale locale)
- DateFormat getDateTimeInstance()
- DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
- DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale locale)
- DateFormat getInstance()
CHAPTER 16: Focusing on Odds and Ends

- DateFormat getTimeInstance()
- DateFormat getTimeInstance(int style)
- DateFormat getTimeInstance(int style, Locale locale)

The getDateInstance() class methods' formatters generate only date information, the getTimeInstance() class methods' formatters generate only time information, and the getDateTimeInstance() class methods' formatters generate date and time information.

The dateStyle and timeStyle fields determine how that information will be presented according to the following DateFormat constants:

- SHORT is completely numeric, such as 12.13.52 or 3:30pm.
- MEDIUM is longer, such as Jan 12, 1952.
- LONG is longer still, such as January 12, 1952 or 3:30:32pm.
- FULL is pretty completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST.

Listing 16-31 shows you how to format a Date instance that represents the Unix epoch according to the local time zone and the UTC time zone.

Listing 16-31. Formatting the Unix Epoch

```java
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class DateFormatDemo
{
    public static void main(String[] args)
    {
        Date d = new Date(0); // Unix epoch
        System.out.println(d);
        DateFormat df = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                      DateFormat.LONG,
                                                      Locale.US);
        System.out.println("Default format: " + df.format(d));
        df.setTimeZone(TimeZone.getTimeZone("UTC"));
        System.out.println("Taking UTC into account: " + df.format(d));
    }
}
```

Compile Listing 16-31 as follows:

`javac DateFormatDemo.java`

Run this application as follows:

`java DateFormatDemo`
I observe the following output for the CST time zone:

<table>
<thead>
<tr>
<th>Date and Time</th>
<th>UTC Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed Dec 31 18:00:00 CST 1969</td>
<td>January 1, 1970 12:00:00 AM UTC</td>
</tr>
</tbody>
</table>

Default format: December 31, 1969 6:00:00 PM CST
Taking UTC into account: January 1, 1970 12:00:00 AM UTC

The Unix epoch, which is represented by passing 0 to the Date(long) constructor, is defined as January 1, 1970 00:00:00 UTC, but the first output line doesn't indicate this fact. Instead, it shows the epoch in my CST time zone, which is six hours away from GMT/UTC. To show the epoch correctly, I need to obtain the UTC time zone, which I accomplish by passing "UTC" to TimeZone's getTimeZone(String) class method and install this time zone instance into the date formatter with the help of DateFormat's void setTimeZone(TimeZone zone) method.

Note: When you need to create customized date formatters, you'll find yourself working with DateFormat's java.text.SimpleDateFormat subclass and this subclass's java.text.DateFormatSymbols companion class. The “Customizing Formats” section (http://download.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html) and the “Changing Date Format Symbols” section (http://download.oracle.com/javase/tutorial/i18n/format/dateFormatSymbols.html) in The Java Tutorials introduce you to these classes.

### Message Formatters

Applications often display simple and/or compound status and error messages to the user. A simple message consists of static (unchanging) text, whereas a compound message consists of static text and variable (changeable) data. For example, consider the following compound messages, where the underlined text identifies variable data:

- 10,536 visitors have visited your website since June 16, 2010.
- Warning: 25 files have been modified in a suspicious manner.
- Account balance is $10,567.00!

For a simple message, you obtain its text from a resource bundle and then display this text to the user. For a compound message, you obtain a pattern (template) for the message from a property resource bundle, pass this pattern along with the variable data to a message formatter to create a simple message, and display this message's text.

A message formatter is an instance of the concrete MessageFormat class (a Format subclass). Unlike other APIs, MessageFormat doesn't have an abstract entry-point class with class methods for obtaining instances of subclasses. Instead, this class declares the following constructors:

- MessageFormat(String pattern) initializes a MessageFormat instance to the specified pattern and the default locale. This constructor throws IllegalArgumentException when pattern is invalid.

- MessageFormat(String pattern, Locale locale) initializes a MessageFormat instance to the specified pattern and locale. This constructor throws IllegalArgumentException when pattern is invalid.
A pattern consists of static text and placeholders for variable data. Each placeholder is a brace-delimited sequence of a zero-based integer identifier, an optional format type, and an optional format style. Examples include \{0\} (insert text between braces), \{1, date\} (insert a date in default style), and \{2, number, currency\} (insert a currency).

For example, the previous set of compound messages can be converted into Listing 16-32’s patterns for the en_US locale.

**Listing 16-32. Patterns in an example_en_US.properties File**

p1 = {0, number, integer} visitors have visited your website since {1, date, long}.
p2 = Warning: {0, number, integer} files have been modified in a suspicious manner.
p3 = Account balance is {0, number, currency}!

The same placeholders can be used in equivalent compound messages localized to another locale, such as Listing 16-33’s fr_FR locale.

**Listing 16-33. Patterns in an example_fr_FR.properties File**

p1 = {0, number, integer} visiteurs ont visité votre site Web depuis le {1, date, long}.
p2 = Avertissement : {0, number, integer} dossiers ont été modifiés d’une façon soupçonneuse.
p3 = L’équilibre de compte est {0, number, currency} !

**Note** An apostrophe (also known as a single quote) in a pattern starts a quoted string, in which, for example, '{0, number, currency} is treated as a literal string and isn’t interpreted as a placeholder by the formatter. To ensure that this placeholder isn’t treated as a literal string in the previous example, L’équilibre’s single quote must be doubled, which is why L''équilibre appears.

After creating a MessageFormat instance, where the pattern is obtained from a resource bundle, you typically create an array of Objects and call MessageFormat’s inherited String format(Object obj) method (from Format) with this array; passing an array of Objects to a method whose parameter type is Object works because arrays are Objects.

When format() is called, it scans the pattern, replacing each placeholder with the corresponding entry in the Objects array. For example, when format() finds a placeholder with integer identifier 0, it causes the zeroth entry in the Objects array to be formatted and then the formatted results to be output.

**Tip** You might find MessageFormat’s String format(String pattern, Object... arguments) class method convenient for one-time formatting operations. This method is equivalent to executing new MessageFormat(pattern).format(arguments, new StringBuffer(), null).toString() on the default locale.
Listing 16-34 demonstrates message formatting in the context of the previous examples’ properties files and their localized patterns.

Listing 16-34. Formatting and Outputting Compound Messages According to the en_US and fr_FR Locales

import java.text.MessageFormat;
import java.util.Calendar;
import java.util.Locale;
import java.util.ResourceBundle;

public class MessageFormatDemo
{
    public static void main(String[] args)
    {
        dumpMessages(Locale.US);
        System.out.println();
        dumpMessages(Locale.FRANCE);
    }

    static void dumpMessages(Locale locale)
    {
        ResourceBundle rb = ResourceBundle.getBundle("example", locale);
        MessageFormat mf = new MessageFormat(rb.getString("p1"), locale);
        Calendar cal = Calendar.getInstance(locale);
        cal.set(Calendar.YEAR, 2010);
        cal.set(Calendar.MONTH, Calendar.JUNE);
        cal.set(Calendar.DAY_OF_MONTH, 16);
        Object[] args = new Object[] { 10536, cal.getTime() };
        System.out.println(mf.format(args));
        mf.applyPattern(rb.getString("p2"));
        args = new Object[] { 25 };
        System.out.println(mf.format(args));
        mf.applyPattern(rb.getString("p3"));
        args = new Object[] { 10567.0 };
        System.out.println(mf.format(args));
    }
}

Listing 16-34 takes advantage of MessageFormat's void applyPattern(String pattern) method to override a previous pattern with a new pattern.

Compile Listing 16-34 as follows:

javac MessageFormatDemo.java

Run this application as follows:

java MessageFormatDemo

You should observe Figure 16-7’s output, which I present via the Windows Notepad application.
If you observe an exception instead of this output, example_en_US.properties and example_fr_FR.properties are probably not in the same directory as MessageFormatDemo.

Note Some compound messages contain singular and plural words. For example, Logging 1 message to x.log and Logging 2 messages to x.log reveal singular and plural messages. Although you could specify pattern Logging {0} message(s) to {1}, it's not grammatically correct to state Logging 2 message(s) to x.log. The solution to this problem is to use the concrete java.text.ChoiceFormat class, a subclass of NumberFormat and a partner of MessageFormat, so that you can output Logging 1 message to x.log or Logging 2 messages to x.log depending on the numeric value passed to {0}. To learn how to use ChoiceFormat, check out the “Handling Plurals” section (http://download.oracle.com/javase/tutorial/i18n/format/choiceFormat.html) in The Java Tutorials.

Parsing

The Format class has a dual personality in that it also declares a pair of parseObject() methods for parsing strings back into objects. Furthermore, it associates with a java.text.ParseException class whose instances are thrown when errors occur during parsing and a java.text.ParsePosition class that keeps track of the current parsing position and error position indexes.

Note ParsePosition declares int getIndex() and void setIndex(int index) methods for getting and setting the current parsing position, and it declares int getErrorIndex() and void setErrorIndex(int index) methods for getting and setting the current error position.
Format declares the following `parseObject()` methods:

- `Object parseObject(String source)` parses `source` from the beginning and returns a corresponding object. Not all of `source`’s text may be parsed. This method throws `ParseException` when the beginning of this text cannot be parsed.

- `Object parseObject(String source, ParsePosition pos)` parses `source` starting at the current parsing position index stored in `pos` and returns a corresponding object. When parsing succeeds, `pos`’s current parsing position index is updated to the index after the last character used (parsing doesn’t necessarily use all characters up to the end of the string), and the parsed object is returned. The updated `pos` can be used to indicate the starting point for the next call to this method. When an error occurs, `pos`’s current parsing position index isn’t changed. However, its error position index is set to the index of the character where the error occurred and null is returned. This method throws `NullPointerException` when null is passed to `pos`.

`parseObject(String)` invokes the abstract `parseObject(String, ParsePosition)` method as if by calling `parseObject(source, new ParsePosition(0))`.

Format subclasses such as `DateFormat` override `parseObject(String, ParsePosition)` to invoke one of their own `parse()` methods. For example, `MessageFormat` overrides `parseObject(String, ParsePosition)` and calls this method when necessary.

Although you should refrain from using specialty subclasses so that your application can adapt to the widest possible audience, you might find occasions to use such subclasses. For example, you might want to work directly with `SimpleDateFormat` to parse legacy date/time strings that were stored in a database according to a specific format. Listing 16-35 demonstrates how you might accomplish this task with help from `SimpleDateFormat`.

**Listing 16-35. Parsing a Date/Time Argument That Must be Formatted According to Specific Date Format**

```java
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class ParseDemo {
    public static void main(String[] args) throws ParseException {
        if (args.length != 1) {
            System.err.println("usage: java ParseDemo yyyy-MM-dd HH:mm:ss z");
            return;
        }
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss z");
        System.out.println(sdf.parse(args[0]));
    }
}
```

www.it-ebooks.info
Listing 16-35 first verifies that a single command-line argument was passed and then instantiates SimpleDateFormat with a pattern string that identifies the pattern to follow for parsing. (The complete details on pattern string syntax are available in SimpleDateFormat’s Java documentation.)

Compile Listing 16-35 as follows:

javac ParseDemo.java

Run this application as follows:

java ParseDemo "2014-01-04 11:21:00 CST"

On my platform, I observe the following output:

```
Sat Jan 04 11:21:00 CST 2014
```

---

**Focusing on Logging**

In Chapter 5, I presented a simple logging framework to demonstrate packages. *Logging* is the recording of data while an application runs and is an important part of application development and maintenance. Messages are *logged* (captured as formatted records) to files or other destinations to help diagnose any problems that arise as an application executes.

Creating your own logging framework is typically a waste of time and you should use the standard java.util.logging package instead. This package implements Java’s Logging API, which was introduced in Java 1.4. Oracle’s documentation for java.util.logging states that there are four main uses for the *logs* (repositories of log entries) generated by the Logging API:

- **Problem diagnosis by end users and system administrators**: Basic information about common problems that can be fixed or tracked locally, such as running out of resources, security failures, and simple configuration errors, is logged for viewing by end users and system administrators.

- **Problem diagnosis by field service engineers**: The logging information used by field service engineers may be considerably more complex and verbose than that required by system administrators. Typically, such information will require extra logging within specific subsystems.

- **Problem diagnosis by the development organization**: When a problem occurs in the field, it may be necessary to return the captured logging information to the original development team for diagnosis. This logging information may be extremely detailed and fairly inscrutable. Such information might include detailed tracing on the internal execution of specific subsystems.

- **Problem diagnosis by developers**: The Logging API also may be used to help debug an application under development. This may include logging information generated by the target application as well as logging information generated by lower-level libraries. While this use is perfectly reasonable, the Logging API isn’t intended to replace the normal debugging and profiling tools that may already exist in the development environment.
Note Android provides the android.util.Log class for logging messages, but you might prefer to use java.util.logging to maintain consistency between your Android and non-Android projects. To configure the Logging API for Android, check out stackoverflow.com's “How to configure java.util.logging on Android?” topic (http://stackoverflow.com/questions/4561345/how-to-configure-java-util-logging-on-android).

Logging API Overview

The java.util.logging package consists of 2 interfaces and 15 classes. This package’s key types are described below:

- **Logger**: This class is the entry-point into the Logging API. It lets you log messages to various destinations such as files or the console.
- **LogRecord**: This class describes a message via a record and is used to pass messages between the logging framework and individual Handlers.
- **Handler**: This class receives messages from Logger and publishes them to a console, a file, a network logging service, and so on.
- **Level**: This class defines a set of standard log levels that can be used to control logging output.
- **Filter**: This interface provides fine-grain control over what is logged, beyond the control provided by log levels.
- **Formatter**: This class provides support for formatting LogRecords into strings.

Figure 16-8 shows the relationships among most of these types.

![Figure 16-8. Relating the logging framework to an application and to the outside world](image-url)

Applications make logging calls on Logger objects, which allocate LogRecord objects that are passed to Handler objects for publication. Loggers and Handlers may use log Levels and (optionally) Filters to decide if they’re interested in a specific LogRecord. When it’s necessary to publish a LogRecord externally, a Handler can use a Formatter to localize and format the message before publishing it to an I/O stream that sends the message to the outside world (the console, a file, or some other destination).
Each logged message is associated with a specific log level, which is an integer that indicates the message’s relative importance. The higher the integer (level), the more important is the message. The Level class declares the following log levels:

- SEVERE: A message level indicating a serious failure. This is the highest value.
- WARNING: A message level indicating a potential problem.
- INFO: A message level for informational messages.
- CONFIG: A message level for static configuration messages.
- FINE: A message level providing tracing information.
- FINER: A fairly detailed tracing message.
- FINEST: A highly detailed tracing message.

Note: There’s also level ALL, which lets you log all records, and level OFF, which lets you turn off logging. It’s also possible to define custom levels, which is beyond the scope of this chapter.

As you move down the list, the amount of logged information increases. It takes time to record all of this information, and accumulated logging time can impact your application’s performance. For this reason, the Logging API defaults to logging only those messages at the Level.INFO level or higher.

### A Hierarchy of Loggers

An application that uses the Logging API typically first obtains a Logger instance (logger). The Logger class declares several getLogger() and getAnonymousLogger() class methods for this task. The getLogger() methods return loggers that are associated with names and the getAnonymousLogger() methods return loggers that have no names.

Loggers are normally named and this name, which typically consists of several dot-separated names (think of a package name), describes a namespace. Each dot-separated name identifies a separate logger in a hierarchy of related loggers. The following example obtains a named logger by calling Logger.getLogger(String name):

```java
Logger logger = Logger.getLogger("simpleLogger");
```

The string passed to getLogger() names the logger, which happens to be simpleLogger in the example.

Note: Logger declares a String getName() method that returns a logger’s name or null for anonymous loggers.
simpleLogger exists in a hierarchy consisting of two loggers:

```
""
simpleLogger
```

Here, "" identifies the root logger, which is an automatically created logger at the top of the hierarchy. The root logger is the parent logger and simpleLogger is the child logger of this parent.

You will often pass a dot-separated name to getLogger(), as follows:

```java
Logger logger = Logger.getLogger("ca.tutortutor.app");
```

The logger associated with ca.tutortutor.app exists in a potential hierarchy of four loggers:

```
""
ca
ca.tutortutor
ca.tutortutor.app
```

At this point, only the root logger and the logger associated with ca.tutortutor.app exist. The other two loggers don’t automatically come into existence and must be explicitly created, as follows:

```java
Logger logger1 = Logger.getLogger("ca");
Logger logger2 = Logger.getLogger("ca.tutortutor");
Logger logger3 = Logger.getLogger("ca.tutortutor.app");
```

What’s the point of having separate loggers? The answer is flexibility. Given these loggers, you could log messages separately based on their locations in the namespace. For example, all messages logged via logger1 could be sent to the console, all messages logged via logger2 could be stored in a file, and so on.

Figure 16-9 shows the resulting hierarchy.

![Diagram of logger hierarchy](image-url)

*Figure 16-9. A hierarchy of loggers*
Logger declares a `Logger getParent()` method that returns the nearest parent of the logger on which this method is invoked. For example, `logger2.getParent()` returns `logger1`. When `getParent()` is called on the root logger, this method returns null. If a logger hasn’t been created, the next highest parent is returned. For example, `Logger.getLogger("a.b").getParent()` returns the root logger.

### Logging Messages

Logger provides methods for sending messages to handlers. By default, a logger also sends its output to its parent logger. Logger declares the following method categories for logging messages:

- **The log() methods** let you log messages at specific log levels. For example, `void log(Level logLevel, String msg)` logs the message identified by `msg` at the specified level. The message is transmitted to all subscribed handlers.

- **The logp() methods** build onto the log() methods by adding `String`-based `sourceClass` and `sourceMethod` parameters, which identify the class and method that was the source of the message. For example, `void logp(Level logLevel, String sourceClass, String sourceMethod, String msg)` is equivalent to the previous log() method except for these additional parameters.

- **The logrb() methods** are also similar to the log() methods. However, they can obtain localized messages from resource bundles, which I discussed earlier in this chapter. For example, `void logrb(Level logLevel, String sourceClass, String sourceMethod, String bundleName, String msg)` is equivalent to the previous logp() method except that a localized variant of `msg` is logged.

- **The miscellaneous category** declares additional methods that include the following methods:
  - `void entering(String sourceClass, String sourceMethod)`: Log a message indicating that a method has been entered. A log record with log level `Level.FINER`, message `ENTRY`, the specified source class name, and the specified source method name is submitted for logging.
  - `void exiting(String sourceClass, String sourceMethod)`: Log a message indicating that a method has exited. A log record with log level `Level.FINER`, message `RETURN`, the specified source class name, and the specified source method name is submitted for logging.
  - `void finest(String msg)`: Log a message of level `Level.FINEST`; the message is transmitted to all subscribed handlers.
  - `void info(String msg)`: Log a message of level `Level.INFO`; the message is transmitted to all subscribed handlers.

Listing 16-36 presents an application that demonstrates `void log(Level logLevel, String msg)`.
Listing 16-36. Logging a Loop of Integers

import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo
{
    public static void main(String[] args)
    {
        Logger logger = Logger.getLogger("LoggingDemo");
        for (int i = 0; i < 5; i++)
            logger.log(Level.INFO, i + "");
    }
}

Listing 16-36’s main() method first obtains a logger named LoggingDemo and then enters a for loop. Each loop iteration invokes the logger’s log() method with the INFO level and the current value of i.

Note It’s common to choose a package name and a class name for the logger’s name. For example, in Listing 16-36, I chose LoggingDemo as this name because it corresponds to the LoggingDemo class name, which exists in the unnamed package (no package statement).

Compile Listing 16-36 as follows:

javac LoggingDemo.java

Run this application as follows:

java LoggingDemo

You should observe output similar to that shown below:

Jan 06, 2014 8:36:08 PM LoggingDemo main
INFO: 0
Jan 06, 2014 8:36:08 PM LoggingDemo main
INFO: 1
Jan 06, 2014 8:36:08 PM LoggingDemo main
INFO: 2
Jan 06, 2014 8:36:08 PM LoggingDemo main
INFO: 3
Jan 06, 2014 8:36:08 PM LoggingDemo main
INFO: 4

This output is sent to the default console handler that forwards the messages to the standard error stream. Notice the word INFO, which indicates that the logged message is an informational message.
Listing 16-37 presents an application that demonstrates `void logp(Level logLevel, String sourceClass, String sourceMethod, String msg)`.

**Listing 16-37. Logging a Loop of Integers Along with Class and Method Names**

```java
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo
{
    public static void main(String[] args)
    {
        Logger logger = Logger.getLogger("LoggingDemo");
        for (int i = 0; i < 5; i++)
            logger.logp(Level.INFO, "LoggingDemo", "main", i+"");
    }
}
```

Compile Listing 16-37 (`javac LoggingDemo.java`) and run the application (`java LoggingDemo`). You should observe output similar to the following output:

```
Jan 06, 2014 8:42:52 PM LoggingDemo main
INFO: 0
Jan 06, 2014 8:42:52 PM LoggingDemo main
INFO: 1
Jan 06, 2014 8:42:52 PM LoggingDemo main
INFO: 2
Jan 06, 2014 8:42:52 PM LoggingDemo main
INFO: 3
Jan 06, 2014 8:42:52 PM LoggingDemo main
INFO: 4
```

Listing 16-38 presents an application that demonstrates `void logrb(Level logLevel, String sourceClass, String sourceMethod, String bundleName, String msg)`.

**Listing 16-38. Logging a Localized France Message Along with Class and Method Names**

```java
import java.util.Locale;
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo
{
    public static void main(String[] args)
    {
        Locale.setDefault(Locale.FRANCE);
        Logger logger = Logger.getLogger("LoggingDemo");
        logger.logrb(Level.INFO, "LoggingDemo", "main", "msg", "hello");
    }
}
```

www.it-ebooks.info
Listing 16-38’s main() method first sets the default locale to Locale.FRANCE. It then obtains a logger and invokes logrb() on this logger to log the localized FRANCE equivalent of the hello message. The equivalent text is stored in a resource bundle named msg_fr_FR.properties, whose contents are presented in Listing 16-39.

**Listing 16-39. Storing FRANCE Locale Equivalent Text for the Hello Message**

```
hello=Bonjour
```

Compile Listing 16-38 (javac LoggingDemo.java) and run the application (java LoggingDemo). You should observe localized output similar to the following output:

```
janv. 06, 2014 11:30:54 PM LoggingDemo main
INFO: Bonjour
```

Listing 16-40 presents an application that demonstrates void entering(String sourceClass, String sourceMethod) and void exiting(String sourceClass, String sourceMethod).

**Listing 16-40. Logging a Recursive factorial() Method’s Entry and Exit**

```
import java.util.logging.Logger;

public class LoggingDemo {
    static Logger logger = Logger.getLogger("LoggingDemo");

    public static void main(String[] args) {
        System.out.println(factorial(5));
    }

    static int factorial(int n) {
        logger.entering("LoggingDemo", "factorial");
        if (n == 0 || n == 1) {
            logger.exiting("LoggingDemo", "factorial");
            return 1;
        } else {
            logger.exiting("LoggingDemo", "factorial");
            return n*factorial(n-1);
        }
    }
}
```

Listing 16-40’s main() method calculates and outputs 5 factorial, which equals 120. This method logs messages upon entry to and exit from the recursive factorial() method.
Compile Listing 16-40 (javac LoggingDemo.java) and run the application (java LoggingDemo). You will probably be surprised to discover no logging messages.

No logging messages are output because the logger defaults to not outputting messages whose log levels are less than Level.INFO, the default console handler doesn’t publish messages whose log levels are less than Level.INFO, and the entering() and exiting() methods log messages whose log levels are set to Level.FINER, which is less than Level.INFO.

The solution to this problem is to change the logger and handler log levels to Level.FINER. Listing 16-41 shows how to accomplish this task.

Listing 16-41. Changing Logger and Handler Levels to Observe Entry and Exit Messages

```java
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo
{
  static Logger logger = Logger.getLogger("LoggingDemo");

  public static void main(String[] args)
  {
    logger.setLevel(Level.FINER);
    Handler[] handlers = Logger.getLogger("").getHandlers();
    for (Handler handler: handlers)
      handler.setLevel(Level.FINER);
    System.out.println(factorial(5));
  }

  static int factorial(int n)
  {
    logger.entering("LoggingDemo", "factorial");
    if (n == 0 || n == 1)
    {
      logger.exiting("LoggingDemo", "factorial");
      return 1;
    }
    else
    {
      logger.exiting("LoggingDemo", "factorial");
      return n*factorial(n-1);
    }
  }
}
```

Listing 16-41 uses Logger's void setLevel(Level newLevel) and Handler[] getHandlers() methods to change respectively the invoking logger's log level to Level.FINER and obtain an array of handlers associated with the root logger (""). For each returned handler (the root logger provides a console handler only), Handler's void setLevel(Level newLevel) method is called to set the handler's log level to Level.FINER.
Compile Listing 16-41 (javac LoggingDemo.java) and run the application (java LoggingDemo). This time, you should observe more complete output:

```
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: ENTRY
Jan 06, 2014 11:32:28 PM LoggingDemo factorial
FINER: RETURN
120
```

The previous example placed entering() and exiting() method calls in a recursive method to log method entry and exit. You can also log thrown exceptions from within a catch block via various Logger methods that take java.lang.Throwable arguments. For example, void log(Level logLevel, String msg, Throwable thrown) is equivalent to void log(Level logLevel, String msg) except that it also lets you log an exception, more specifically, a throwable (an exception or error). Listing 16-42 demonstrates this method.

**Listing 16-42. Logging a Thrown Exception**

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java LoggingDemo filespec");
            return;
        }
```
Listing 16-42’s main() method first verifies that a single command-line argument has been specified. This argument names a file to be opened. After creating a logger, main() attempts to open this file. If java.io.FileNotFoundException is thrown, this exception is logged from within the catch block.

Compile Listing 16-42 (javac LoggingDemo.java), and run the application with a nonexistent file, as follows:

```shell
java LoggingDemo x.dat
```

You should observe output that’s similar to the following output:

```plaintext
Jan 07, 2014 11:04:52 AM LoggingDemo main
INFO: file not found
java.io.FileNotFoundException: x.dat (The system cannot find the file specified)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)
    at java.io.FileInputStream.<init>(FileInputStream.java:97)
    at LoggingDemo.main(LoggingDemo.java:21)
```

**Filtering LogRecords**

Log levels are a kind of filter that let you categorize messages according to their level of importance and the amount of detail to be presented. The Logging API also provides a more generic filtering mechanism based on the Filter interface that lets you further filter messages regardless of their log levels.
Filter declares a single method:

```java
boolean isLoggable(LogRecord record)
```

The `isLoggable()` method receives a `LogRecord` argument, examines this argument to determine whether or not it should be logged, and returns true when the record should be logged or false when the record shouldn’t be logged. Creating a filter is simply a matter of declaring a class that implements `Filter` and its `isLoggable()` method.

`LogRecord` declares various getters and setters for accessing and modifying the record. You will typically only use getters to interrogate the record to learn whether it should be logged or not.

A few useful getters are described here:

- `Level getLevel()`: Returns the record’s log level.
- `String getLoggerName()`: Returns the record’s logger name.
- `String getMessage()`: Returns the record’s raw message, which might be null.
- `long getMillis()`: Returns the time (in milliseconds since 1970) when this record was created.
- `String getSourceMethodName()`: Returns the record’s name of the method that’s the source of this record.

Listing 16-43 declares a `SimpleFilter` class that uses `getSourceMethodName()` to detect the `main()` method and accepts records originating from this method only.

```java
Listing 16-43. Declaring a Filter That Accepts Records from the main() Method Only
```

```java
import java.util.logging.Filter;
import java.util.logging.LogRecord;

public class SimpleFilter implements Filter {
    @Override
    public boolean isLoggable(LogRecord record) {
        return (record.getSourceMethodName().equals("main")) ? true : false;
    }
}
```

Logger declares a `void setFilter(Filter newFilter)` method for installing a filter on the current logger. Passing null to `newFilter` uninstalls the filter. A companion `Filter getFilter()` method returns the current filter or null when there is none.

**Note** You can also install filters on handlers. `Handler` declares identical `void setFilter(Filter newFilter)` and `Filter getFilter()` methods.
Listing 16-44 declares a LoggingDemo class that demonstrates the use of SimpleFilter.

Listing 16-44. Filtering Records from main() and a Companion Method

```java
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger("LoggingDemo");
        logger.setFilter(new SimpleFilter());
        logger.log(Level.INFO, "Message 1");
        logger.log(Level.INFO, "Message 2");
        foo(logger);
        System.out.println();
        logger.setFilter(null);
        foo(logger);
    }

    static void foo(Logger logger) {
        logger.log(Level.INFO, "Message 3");
    }
}
```

Listing 16-44’s main() method first obtains a logger and installs an instance of SimpleFilter on this logger. It then logs a couple of informational messages and invokes a companion foo() method with the logger as its argument; this method logs a third informational message. The filter is then removed and foo() is called one more time.

Compile Listing 16-44 (javac LoggingDemo.java), which also compiles SimpleFilter.java (assuming that it's located in the same directory), and run the application (java LoggingDemo). You should observe output that’s similar to the following output:

```
Jan 07, 2014 12:08:26 PM LoggingDemo main
INFO: Message 1
Jan 07, 2014 12:08:26 PM LoggingDemo main
INFO: Message 2

Jan 07, 2014 12:08:26 PM LoggingDemo foo
INFO: Message 3
```

The message logged in the foo() method isn’t written to the console when the SimpleFilter instance is installed. It's written only when this filter is removed.
Handlers and Formatters

Loggers send log records to handlers, which are instances of classes that extend the abstract `Handler` class. These handler instances publish records to various destinations when their concrete implementations of `Handler`’s `void publish(LogRecord record)` method are called by the logging framework.

The Logging API provides several concrete implementations of the `Handler` class:

- `ConsoleHandler`: A handler that writes log records to the standard error stream.
- `FileHandler`: A handler that writes log records to a file or to a rotating set of files.
- `MemoryHandler`: A handler that writes log records to a cycled memory buffer.
- `SocketHandler`: A handler that writes log records to a socket.
- `StreamHandler`: A handler that writes log records to an output stream.

**Note** If none of these handlers meet your needs, you can always extend `Handler` and define your own handler class.

Every logger has at least a default console handler. You can add more handlers to, remove existing handlers from, and query all currently registered handlers on a logger by invoking the following `Logger` methods:

- `void addHandler(Handler handler)`: Adds the specified handler to the invoking logger.
- `void removeHandler(Handler handler)`: Removes the specified handler from the invoking logger. This method returns silently when the handler isn’t found or null is passed.
- `Handler[] getHandlers()`: Returns an array of all handlers registered with the invoking logger.

Listing 16-45 extends Listing 16-36 by also adding a file handler as a destination for logged messages.

**Listing 16-45. Logging Messages to the Standard Error Stream and to a File**

```java
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```java
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
```
public class LoggingDemo
{
    public static void main(String[] args) throws IOException
    {
        Logger logger = Logger.getLogger("LoggingDemo");
        logger.addHandler(new FileHandler("log"));
        for (int i = 0; i < 5; i++)
            logger.log(Level.INFO, i++);
    }
}

Listing 16-45's `main()` method instantiates `FileHandler` by invoking its `FileHandler(String pattern)` constructor. The argument passed to `pattern` is the name of the file to which logged messages are written. Because this constructor can throw `IOException`, I've appended a `throws IOException` clause to `main()`'s header.

Continuing, the file handler is added to the logger and various messages are logged.

Note: `FileHandler` declares a `void close()` method for closing all files associated with the file handler. You would typically invoke this method after logging to the file handler in a long-running application. For a trivial application, such as that shown in Listing 16-45 in which the file is closed when the application exits, it isn't essential to call `close()`.

Compile Listing 16-45 `(javac LoggingDemo.java)` and run the application `(java LoggingDemo)`. In addition to observing logging information on the console, you should also observe the creation of a file named `log` that stores the output. This file's content contains the following:

```xml
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
    <record>
        <date>2014-01-07T13:13:16</date>
        <millis>1389121996785</millis>
        <sequence>0</sequence>
        <logger>LoggingDemo</logger>
        <level>INFO</level>
        <class>LoggingDemo</class>
        <method>main</method>
        <thread>1</thread>
        <message>0</message>
    </record>
    <record>
        <date>2014-01-07T13:13:16</date>
        <millis>1389121996815</millis>
        <sequence>1</sequence>
        <logger>LoggingDemo</logger>
        <level>INFO</level>
```
<class>LoggingDemo</class>
<method>main</method>
<thread>1</thread>
<message>1</message>
</record>
<record>
<date>2014-01-07T13:13:16</date>
<millis>1389121996825</millis>
<sequence>2</sequence>
<logger>LoggingDemo</logger>
<level>INFO</level>
<class>LoggingDemo</class>
<method>main</method>
<thread>1</thread>
<message>2</message>
</record>
<record>
<date>2014-01-07T13:13:16</date>
<millis>1389121996825</millis>
<sequence>3</sequence>
<logger>LoggingDemo</logger>
<level>INFO</level>
<class>LoggingDemo</class>
<method>main</method>
<thread>1</thread>
<message>3</message>
</record>
<record>
<date>2014-01-07T13:13:16</date>
<millis>1389121996825</millis>
<sequence>4</sequence>
<logger>LoggingDemo</logger>
<level>INFO</level>
<class>LoggingDemo</class>
<method>main</method>
<thread>1</thread>
<message>4</message>
</record>
</log>

This formatted output was created by an instance of the XMLFormatter class, which is a concrete subclass of its abstract Formatter superclass. Formatter objects convert LogRecords to string representations, and XMLFormatter converts LogRecords to XML string representations.

Note The java.util.logging package also offers SimpleFormatter as a concrete Formatter subclass.
Formatters are associated with handlers. Handler declares void setFormatter(Formatter newFormatter) for setting the invoking handler’s formatter to newFormatter and a companion Formatter getFormatter() method for returning the current formatter.

You can use this knowledge to change the formatter assigned to the previous FileHandler object from an XML formatter to a simple formatter. Listing 16-46 demonstrates this change.

Listing 16-46. Logging Simple Formatted Messages to the Standard Error Stream and to a File

```java
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;
public class LoggingDemo
{
    public static void main(String[] args) throws IOException
    {
        Logger logger = Logger.getLogger("LoggingDemo");
        FileHandler fh = new FileHandler("log");
        fh.setFormatter(new SimpleFormatter());
        logger.addHandler(fh);
        for (int i = 0; i < 5; i++)
            logger.log(Level.INFO, i+"");
    }
}
```

Compile Listing 16-46 (javac LoggingDemo.java) and run the application (java LoggingDemo). In addition to observing logging information on the console, you should also observe the creation of a file named log that stores the output. This file’s simple formatted content appears as follows:

```
Jan 07, 2014 1:22:05 PM LoggingDemo main
INFO: 0
Jan 07, 2014 1:22:05 PM LoggingDemo main
INFO: 1
Jan 07, 2014 1:22:05 PM LoggingDemo main
INFO: 2
Jan 07, 2014 1:22:05 PM LoggingDemo main
INFO: 3
Jan 07, 2014 1:22:05 PM LoggingDemo main
INFO: 4
```

LogManager and Configuration

The java.util.logging package includes a LogManager class that’s used to manage a hierarchical namespace of all named Logger objects and to maintain configuration properties of the logging framework. Although you will typically not need to work with LogManager, you may want to do so to reload configuration information or to perform another global task.
The virtual machine includes a single log manager, which you obtain by invoking LogManager's LogManager.getLogManager() class method, as follows:

    LogManager lm = LogManager.getLogManager();

You can then invoke a LogManager method such as void readConfiguration() to reinitialize the log manager's properties and configuration.

The Logging API has a default configuration file named logging.properties that's stored in the JDK's %JAVA_HOME%/jre/lib directory. This properties file provides default settings such as the default global log level and the default log level for all console handlers. Listing 16-47 presents this file's contents for JDK 7 Update 6.

Listing 16-47. The Default Logging Configuration File

```
# Default Logging Configuration File
#
# You can use a different file by specifying a filename
# with the java.util.logging.config.file system property.
# For example java -Djava.util.logging.config.file=myfile

# Global properties

# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overridden by a facility specific level
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.
.level= INFO

# Handler specific properties.
# Describes specific configuration info for Handlers.

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
```
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Example to customize the SimpleFormatter output format
# to print one-line log message like this:
#     <level>: <log message> [<date/time>]
#
# java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n

############################################################
# Facility specific properties.
# Provides extra control for each logger.
############################################################

# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
com.xyz.foo.level = SEVERE

You might prefer to change the default configuration. Instead of modifying the JDK’s
logging.properties file, you should create a separate configuration file and set the virtual machine’s
java.util.logging.config.file system property to point to this file.

For example, let’s create a mylogging.properties file that’s identical to logging.properties with two
exceptions:

- The global logging level is changed to FINEST by replacing .level= INFO with
  .level=FINEST.
- The console handler level is changed to FINEST by replacing java.util.logging.
  ConsoleHandler.level = INFO with java.util.logging.ConsoleHandler.level =
  FINEST.

Also, you need a LoggingDemo application that logs its messages using the FINEST log level.
Listing 16-48 presents this application.

Listing 16-48. Logging a Loop of Integers Revisited

import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingDemo
{
    public static void main(String[] args)
    {
        Logger logger = Logger.getLogger("LoggingDemo");
        for (int i = 0; i < 5; i++)
            logger.log(Level.FINE, i + "");
    }
}
Compile Listing 16-48 (javac LoggingDemo.java), and run the application (java LoggingDemo). You should not observe any output.

Now, re-execute the application via the following command line:

```
java -Djava.util.logging.config.file=mylogging.properties LoggingDemo
```

This time, you should observe the following output:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Class</th>
<th>Method</th>
<th>Message</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 07, 2014 3:48:27 PM</td>
<td>LoggingDemo main</td>
<td>FINE: 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan 07, 2014 3:48:27 PM</td>
<td>LoggingDemo main</td>
<td>FINE: 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan 07, 2014 3:48:27 PM</td>
<td>LoggingDemo main</td>
<td>FINE: 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan 07, 2014 3:48:27 PM</td>
<td>LoggingDemo main</td>
<td>FINE: 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan 07, 2014 3:48:27 PM</td>
<td>LoggingDemo main</td>
<td>FINE: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ErrorManager**

The Logging API's logging methods never throw exceptions; it would be burdensome to force developers to wrap all of their logging calls in try/catch constructs. Besides, how would the application recover from an exception in the logging framework? Because Handler classes such as FileHandler and StreamHandler can experience I/O exceptions, the Logging API provides the ErrorManager class so that a handler can report an exception instead of discarding it.

Each Handler subclass instance has an associated ErrorManager/ErrorManager subclass instance. When an exception occurs in the handler, it passes the exception as well as a message and an error code (discussed shortly) to the following method:

```java
void error(String msg, Exception ex, int code)
```

The argument passed to `msg` is a descriptive string of the error; it may be `null`. The argument passed to `ex` identifies the exception that was thrown; it may be `null`. Finally, the argument passed to `code` is one of the following error code constants declared by `ErrorManager`:

- `CLOSE_FAILURE`: An output stream couldn’t be closed.
- `FLUSH_FAILURE`: An output stream couldn’t be flushed.
- `FORMAT_FAILURE`: A formatting operation failed.
- `GENERIC_FAILURE`: The failure doesn’t fit into any of the other categories.
- `OPEN_FAILURE`: An output stream couldn’t be opened.
- `WRITE_FAILURE`: An output stream couldn’t be written.
ErrorManager's error() method writes a message that describes the exception to System.err. However, it only does this the first time it's called. The logging framework assumes that a handler that throws an exception will continue to throw the same exception with each subsequently logged message. It's not useful to keep sending repeated exception messages to System.err.

Listing 16-49 presents a LoggingDemo application that demonstrates the default error manager.

Listing 16-49. Generating an Error by Prematurely Closing the Output Stream

```java
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.ErrorManager;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;
import java.util.logging.StreamHandler;

public class LoggingDemo {
    public static void main(String[] args) throws IOException {
        Logger logger = Logger.getLogger("LoggingDemo");
        FileOutputStream fos = new FileOutputStream("log");
        StreamHandler sh = new StreamHandler(fos, new SimpleFormatter());
        logger.addHandler(sh);
        for (int i = 0; i < 5; i++) {
            logger.log(Level.INFO, i + "");
            if (i == 0)
                fos.close();
        }
    }
}
```

Listing 16-49's main() method first obtains a logger and then creates an output stream to a file named log. Next, it creates a StreamHandler instance connected to this file output stream and a simple formatter. The stream handler is then added to the logger.

main() now enters the for loop and logs an informational message to the output stream. After logging this message, it closes the output stream and starts another for loop iteration. The next call to the log() method results in an error, specifically a thrown exception.
Compile Listing 16-49 (javac LoggingDemo.java) and run the application (java LoggingDemo).
You should observe the following output:

Jan 07, 2014 11:19:48 PM LoggingDemo main
INFO: 0
Jan 07, 2014 11:19:48 PM LoggingDemo main
INFO: 1
Jan 07, 2014 11:19:48 PM LoggingDemo main
INFO: 2
Jan 07, 2014 11:19:48 PM LoggingDemo main
INFO: 3
Jan 07, 2014 11:19:48 PM LoggingDemo main
INFO: 4
java.util.logging.ErrorManager: 3
java.io.IOException: Stream Closed
at java.io.FileOutputStream.writeBytes(Native Method)
    at java.io.FileOutputStream.write(FileOutputStream.java:318)
    at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:221)
    at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:291)
    at sun.nio.cs.StreamEncoder.implFlush(StreamEncoder.java:295)
    at sun.nio.cs.StreamEncoder.flush(StreamEncoder.java:141)
    at java.io.OutputStreamWriter.flush(OutputStreamWriter.java:229)
    at java.util.logging.StreamHandler.flushAndClose(StreamHandler.java:260)
    at java.util.logging.StreamHandler.close(StreamHandler.java:284)
    at java.util.logging.LogManager.resetLogger(LogManager.java:860)
    at java.util.logging.LogManager.reset(LogManager.java:843)
    at java.util.logging.LogManager$Cleaner.run(LogManager.java:240)

The default error manager identifies the cause of the thrown IOException, which happens to be that the output stream was prematurely closed while logging was still in progress.

You can subclass ErrorManager to customize the error reporting and install it on the handler by passing an instance to Handler's void setErrorManager(ErrorManager em) method. A companion ErrorManager getErrorManager() method returns the current error manager.

Listing 16-50 extends Listing 16-49 by installing a custom error manager.

Listing 16-50. Customized Error Output from a Prematurely Closed Output Stream

import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.ErrorManager;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;
import java.util.logging.StreamHandler;
public class LoggingDemo
{
    public static void main(String[] args) throws IOException
    {
        Logger logger = Logger.getLogger("LoggingDemo");
        FileOutputStream fos = new FileOutputStream("log");
        StreamHandler sh = new StreamHandler(fos, new SimpleFormatter());
        sh.setErrorManager(new MyErrorManager());
        logger.addHandler(sh);
        for (int i = 0; i < 5; i++)
        {
            logger.log(Level.INFO, i+"");
            if (i == 0)
            {
                fos.close();
            }
        }
    }
}

class MyErrorManager extends ErrorManager
{
    @Override
    public void error(String msg, Exception ex, int code)
    {
        System.err.println("===============================================");
        super.error(msg, ex, code);
        System.err.println("===============================================");
    }
}

Compile Listing 16-50 (javac LoggingDemo.java) and run the application (java LoggingDemo). You should observe the following output:

Jan 08, 2014 7:14:14 PM LoggingDemo main
INFO: 0
Jan 08, 2014 7:14:14 PM LoggingDemo main
INFO: 1
Jan 08, 2014 7:14:14 PM LoggingDemo main
INFO: 2
Jan 08, 2014 7:14:14 PM LoggingDemo main
INFO: 3
Jan 08, 2014 7:14:14 PM LoggingDemo main
INFO: 4
===============================================
java.util.logging.ErrorManager: 3
java.io.IOException: Stream Closed
    at java.io.FileOutputStream.writeBytes(Native Method)
    at java.io.FileOutputStream.write(FileOutputStream.java:318)
    at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:221)
    at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:291)
    at sun.nio.cs.StreamEncoder.implFlush(StreamEncoder.java:141)
Focusing on Preferences

Significant applications have preferences, which are configuration items. Examples include the location and size of the application’s main window and the locations and names of files that the application most recently accessed. Preferences are persisted to a file, to a database, or to some other storage mechanism so that they will be available to the application the next time it runs.

The simplest approach to persisting preferences is to use the Properties API, which consists of the Properties class. This class persists preferences as a series of key=value entries to text-based properties files. Although properties files are ideal for simple applications with few preferences, they have proven to be problematic with larger applications:

- Properties files tend to grow in size, and the probability of name collisions among the various keys increases. This problem could be eliminated if properties files stored preferences in a hierarchy, but they’re nonhierarchical.

- As an application grows in size and complexity, it tends to acquire numerous properties files with each part of the application associated with its own properties file. The names and locations of these properties files must be hard-coded in the application’s source code.

Additionally, someone could directly modify these text-based properties files (perhaps inserting gibberish), causing the application that depends upon the modified properties file to crash unless it’s properly coded to deal with this possibility. Also, properties files cannot be used on diskless computing platforms. Because of these problems, Java offers the Preferences API as a replacement for the Properties API.

Note Android provides its own preferences mechanism that you’ll most likely use when creating Android apps. However, it’s still good to know about Preferences. For example, suppose your company plans to develop an app that must communicate with a server-based application that you must develop. This application may need to use the Preferences API to persist its preferences.
Exploring Preferences

The Preferences API lets you store preferences in a hierarchical manner so that you can avoid name collisions. Because this API is backend-neutral, it doesn’t matter where the preferences are stored (a file, a database, or [on Windows platforms] the registry); you don’t have to hardcode filenames and locations. Also, there are no text files that can be modified, and Preferences can be used on diskless platforms.

This API uses trees of nodes to manage preferences. These nodes are the analogue of a hierarchical filesystem’s directories. Also, preference name/value pairs stored under these nodes are the analogues of a directory’s files. You navigate these trees in a similar manner to navigating a filesystem: specify an absolute path starting from the root node (/) to the node of interest; for example, /window/location and /window/size.

There are two kinds of trees: system and user. All users share the system preference tree, whereas the user preference tree is specific to a single user, which is generally the person who logged into the underlying platform. (The precise description of “user” and “system” varies from implementation to implementation of the Preferences API.)

Although the Preferences API’s java.util.prefs package contains three interfaces (NodeChangeListener, PreferencesChangeListener, and PreferencesFactory), four regular classes (AbstractPreferences, NodeChangeEvent, PreferenceChangeEvent, and Preferences), and two exception classes (BackingStoreException and InvalidPreferencesFormatException), you mostly work with the Preferences class.

The Preferences class describes a node in a tree of nodes. To obtain a Preferences node, you must call one of the following class methods:

- Preferences systemNodeForPackage(Class<?> c): Returns the node whose path corresponds to the package containing the class represented by c from the system preference tree.
- Preferences systemRoot(): Returns the root preference node of the system preference tree.
- Preferences userNodeForPackage(Class<?> c): Returns the node whose path corresponds to the package containing the class represented by c from the current user’s preference tree.
- Preferences userRoot(): Returns the root preference node of the current user’s preference tree.

Listing 16-51 demonstrates systemNodeForPackage() along with Preferences’ void put(String key, String value) and String get(String key, String def) methods for storing String-based preferences to and retrieving String-based preferences from the system preference tree. A default value must be passed to get() in case no value is associated with the key (which shouldn’t happen in this example).
Listing 16-51. Storing a Single Preference to and Retrieving a Single Preference from the System Preference Tree

```java
package ca.tutortutor.examples;

import java.util.prefs.Preferences;

public class PrefsDemo
{
    public static void main(String[] args)
    {
        Preferences prefs = Preferences.systemNodeForPackage(PrefsDemo.class);
        prefs.put("version", "1.0");
        System.out.println(prefs.get("version", "unknown"));
    }
}
```

Create a `ca\tutortutor\examples` (or `ca/tutortutor/examples`) directory hierarchy under the current directory, and copy `PrefsDemo.java` into `examples`. Assuming that you haven’t changed the current directory to another directory, execute either of the following commands to compile this source file:

```
javac ca\tutortutor\examples\PrefsDemo.java
javac ca/tutortutor/examples/PrefsDemo.java
```

Run this application as follows:

```
java ca.tutortutor.examples.PrefsDemo
```

You should observe the following output:

```
1.0
```

However, if you run this application on Windows 7, you might encounter the following warning messages followed by unknown instead:

```
Jan 08, 2014 12:23:22 PM java.util.prefs.WindowsPreferences <init>
WARNING: Could not create windows registry node Software\JavaSoft\Prefs\ca at root 0x80000002. Windows RegCreateKeyEx(...) returned error code 5.
Jan 08, 2014 12:23:23 PM java.util.prefs.WindowsPreferences WindowsRegOpenKey1
WARNING: Trying to recreate Windows registry node Software\JavaSoft\Prefs\ca at root 0x80000002.
Jan 08, 2014 12:23:23 PM java.util.prefs.WindowsPreferences openKey
WARNING: Could not open windows registry node Software\JavaSoft\Prefs\ca at root 0x80000002. Windows RegOpenKey(...) returned error code 2.
Jan 08, 2014 12:23:23 PM java.util.prefs.WindowsPreferences WindowsRegOpenKey1
WARNING: Trying to recreate Windows registry node Software\JavaSoft\Prefs\ca\tutortutor at root 0x80000002.
Jan 08, 2014 12:23:23 PM java.util.prefs.WindowsPreferences openKey
WARNING: Could not open windows registry node Software\JavaSoft\Prefs\ca\tutortutor at root 0x80000002. Windows RegOpenKey(...) returned error code 2.
Jan 08, 2014 12:23:23 PM java.util.prefs.WindowsPreferences WindowsRegOpenKey1
```

www.it-ebooks.info
These warning messages have to do with Windows security and its User Access Control (UAC) system. You need to disable UAC before you can run this program because it’s making system-oriented changes. You can learn how to disable UAC by checking out Microsoft’s instructions at http://windows.microsoft.com/en-us/windows/turn-user-account-control-on-off#1TC=windows-7.

Assuming that you can view the aforementioned 1.0, you should also be able to see some changes in the Windows 7 registry. Figure 16-10 reveals these changes.

**Figure 16-10. The Windows 7 registry reveals the hierarchy for accessing the version key**

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Prefs is the path to the Windows 7 registry area for storing system preferences. Under Prefs, you’ll find a node for each package stored in this area. For example, ca identifies Listing 16-51’s ca package. Continuing, a hierarchy of nodes is stored under ca. Within the bottommost node (examples), you find an entry consisting of version (the key) and 1.0 (the value).

Listing 16-52 provides a second example that works with the user preference tree and presents a more complex key.
Listing 16-52. Storing a Single Preference to and Retrieving a Single Preference from the Current User’s Preference Tree

```java
package ca.tutortutor.examples;
import java.util.prefs.Preferences;
public class PrefsDemo {
    public static void main(String[] args) {
        Preferences prefs = Preferences.userNodeForPackage(PrefsDemo.class);
        prefs.put("SearchEngineURL", "http://www.google.com");
        System.out.println(prefs.get("SearchEngineURL", "http://www.bing.com"));
    }
}
```

Compile this listing and run the application as previously demonstrated. You shouldn’t observe any security-oriented warning messages. Instead, you should observe the following output:

```
http://www.google.com
```

More interestingly, Figure 16-11 reveals how this preference is stored in the Windows 7 registry.

![Registry Editor](image)

**Figure 16-11. The Windows 7 registry reveals the hierarchy for accessing the SearchEngineURL key**

Computer\HKEY_CURRENT_USER\Software\JavaSoft\Prefs\ca\tutortutor\examples is the path to the Windows 7 registry area for storing user preferences. The Windows 7 registry encodes the SearchEngineURL key into /Search/Engine/U/R/L, because Preferences regards keys and node names as case sensitive but the Windows 7 registry doesn’t.
Focusing on Runtime and Process

The `java.lang.Runtime` class provides Java applications with access to their runtime environment. An instance of this class is obtained by invoking its `Runtime.getRuntime()` class method.

There is only one instance of the `Runtime` class.

Runtime declares several methods that are also declared in `System`. For example, `Runtime` declares a `void gc()` method. Behind the scenes, `System` defers to its `Runtime` counterpart by first obtaining the `Runtime` instance and then invoking this method via that instance. For example, `System`'s static `void gc()` method executes `Runtime.getRuntime().gc()`.

Runtime also declares methods with no `System` counterparts. The following list describes a few of these methods:

- `int availableProcessors()` returns the number of processors that are available to the virtual machine. The minimum value returned by this method is 1.
- `long freeMemory()` returns the amount of free memory (measured in bytes) that the virtual machine makes available to the application.
- `long maxMemory()` returns the maximum amount of memory (measured in bytes) that the virtual machine may use (or `java.lang.Long.MAX_VALUE` when there is no limit).
- `long totalMemory()` returns the total amount of memory (measured in bytes) that is available to the virtual machine. This amount may vary over time depending on the environment that is hosting the virtual machine.

Listing 16-53 demonstrates these methods.

**Listing 16-53. Experimenting with Runtime Methods**

```java
public class RuntimeDemo {
    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("Available processors: "+ rt.availableProcessors());
        System.out.println("Free memory: "+ rt.freeMemory());
    }
}
```
CHAPTER 16: Focusing on Odds and Ends

System.out.println("Maximum memory: " + rt.maxMemory());
System.out.println("Total memory: " + rt.totalMemory());
}

Compile Listing 16-53 as follows:

javac RuntimeDemo.java

Run this application as follows:

java RuntimeDemo

When I run this application on my platform, I observe the following results:

Available processors: 2
Free memory: 123997936
Maximum memory: 1849229312
Total memory: 124649472

Some of Runtime’s methods are dedicated to executing other applications. For example, Process
exec(String program) executes the program named program in a separate native process (executing
application). The new process inherits the environment of the method’s caller, and a java.lang.
Process object is returned to allow communication with the new process. IOException is thrown
when an I/O error occurs.

Tip The java.lang.ProcessBuilder class is a convenient alternative for configuring process attributes
and running a process. For example, Process p = new ProcessBuilder("myCommand",
"myArg").start();

Table 16-3 describes Process methods.

Table 16-3. Process Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void destroy()</td>
<td>Terminates the calling process and closes any associated streams.</td>
</tr>
</tbody>
</table>
| int exitValue()   | Returns the exit value of the native process represented by this Process object (the new process). java.lang.
|                   | IllegalThreadStateException is thrown when the native process has not yet terminated. |

(continued)
Table 16-3. (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputStream getErrorStream()</td>
<td>Returns an input stream that's connected to the standard error stream of the native process represented by this Process object. The stream obtains data piped from the error output of the process represented by this Process object.</td>
</tr>
<tr>
<td>InputStream getInputStream()</td>
<td>Returns an input stream that's connected to the standard output stream of the native process represented by this Process object. The stream obtains data piped from the standard output of the process represented by this Process object.</td>
</tr>
<tr>
<td>OutputStream getOutputStream()</td>
<td>Returns an output stream that's connected to the standard input stream of the native process represented by this Process object. Output to the stream is piped into the standard input of the process represented by this Process object.</td>
</tr>
<tr>
<td>int waitFor()</td>
<td>Causes the calling thread to wait for the native process associated with this Process object to terminate. The process's exit value is returned. By convention, 0 indicates normal termination. This method throws java.lang.InterruptedException when the current thread is interrupted by another thread while waiting.</td>
</tr>
</tbody>
</table>

Listing 16-54 demonstrates exec(String program) and three of Process's methods.

Listing 16-54. Executing Another Application and Displaying Its Standard Output/Error Content

```java
import java.io.InputStream;
import java.io.IOException;

public class Exec
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java Exec program");
            return;
        }
        try
        {
            Process p = Runtime.getRuntime().exec(args[0]);
            // Obtaining process standard output.
            InputStream is = p.getInputStream();
            int _byte;
            while ((_byte = is.read()) != -1)
            {
                System.out.print((char) _byte);
            }
        }
    }
}
```

www.it-ebooks.info
After verifying that exactly one command-line argument has been specified, Listing 16-54’s main() method attempts to run the application identified by this argument. IOException is thrown when the application cannot be located or when some other I/O error occurs.

Assuming that everything is fine, getInputStream() is called to obtain a reference to an input stream that’s used to input the bytes that the newly invoked application writes to its standard output stream, if any. These bytes are subsequently output.

Next, main() calls getErrorStream() to obtain a reference to an input stream that’s used to input the bytes that the newly invoked application writes to its standard error stream, if any. These bytes are subsequently output.

Note To guard against confusion, remember that Process’s getInputStream() method is used to read bytes that the new process writes to its output stream, whereas Process’s getErrorStream() method is used to read bytes that the new process writes to its error stream.

Finally, main() calls waitFor() to block until the new process exits. If the new process is a GUI-based application, this method won’t return until you explicitly terminate the new process. For simple command-line-based applications, Exec should terminate immediately.

Compile Listing 16-54 as follows:

javac Exec.java

Then execute a command line that identifies an application, such as the java application launcher:

java Exec java

You should observe java’s usage information followed by the following line:

Exit status: 1
Caution Because some native platforms provide limited buffer size for standard input and output streams, failure to promptly write the new process’s input stream or read its output stream may cause the new process to block, or even deadlock.

Focusing on the Java Native Interface

The Java Native Interface (JNI) is a native programming interface that lets Java code running in a virtual machine interoperate with native libraries (platform-specific libraries) written in other languages (such as C, C++, or even assembly language). The JNI is a bridge between the virtual machine and the underlying platform to which the native libraries target.

Note The standard class library uses the JNI to read/write files, output graphics to the screen, send data to a network socket, and so on. These tasks are platform-specific and are performed by the underlying platform.

Your applications can leverage the JNI when necessary. There are three common scenarios where you would leverage the JNI to bypass the virtual machine:

- The application must access an unsupported platform-specific API (such as the Win32/Win64 Joystick [http://en.wikipedia.org/wiki/Joystick] API).
- The application must interact with a native library of legacy code (such as a library of statistics routines that predates Java).
- The application must perform complex calculations that exhibit better performance in native code (platform-specific instructions) than in Java code.

You would also use the JNI in an Android context when working with Android’s Native Development Kit (NDK).

Note The decision to use the JNI shouldn’t be made lightly. This technology restricts application execution to specific platforms (such as Windows only), is complex (subtle errors can crash the virtual machine), requires you to learn how to use the C language (or its C++ offspring), and causes application performance to suffer when native code is called frequently (native calls are often 2-3 times slower than Java calls).

Creating a Hybrid Library

When working with the JNI, you’re essentially creating a hybrid library consisting of a Java class and a native interface library. This is explained in the following sections.
Java Class

The Java class uses the native reserved word to declare one or more native methods (methods whose bodies consist of native code stored in a native interface library). Consider the following example:

```java
static native int joyGetNumDevs();
```

This example declares a native `joyGetNumDevs()` class method for returning the number of joystick devices (as a 32-bit integer) that are supported by the underlying Windows platform's joystick driver. When the Java compiler encounters native, it marks this method as a native method in the classfile. At runtime, the virtual machine knows that it must transfer execution to the equivalent native code when it encounters a call to `joyGetNumDevs()`.

The Java class also provides code to load the native interface library that provides a native function (a C-based function) counterpart for each declared native method. This code is often specified via a class initializer, as follows:

```java
static {
    System.loadLibrary("joystick");
}
```

This example's class initializer invokes `System's void loadLibrary(String libname)` class method, where `libname` is the name of the native library without a platform-specific prefix (such as `lib`) or extension (such as `.so` or `.dll`). This method either loads the native library or throws an exception: `NullPointerException` is thrown when you pass null to `libname` and `java.lang.UnsatisfiedLinkError` is thrown when the library doesn’t exist (or doesn’t contain the equivalent native function).

Because of the possibility for `UnsatisfiedLinkError`, which causes the application to terminate when thrown from a class initializer, I've often found it convenient to express the aforementioned code via an `init()` class method that returns a Boolean true/false value.

Listing 16-55 presents a Joystick class that declares the `joyGetNumDevs()` native method and the `init()` class method.

**Listing 16-55. The Java Side of the Joystick Hybrid Library**

```java
class Joystick {
    static boolean init() {
        try {
            System.loadLibrary("joystick");
            return true;
        }
    }
}
```
catch (UnsatisfiedLinkError ule)
{
    return false;
}

static native int joyGetNumDevs();

Listing 16-55’s init() method returns true when the native interface library is successfully loaded; otherwise, it returns false. You can call this method at application startup and gracefully degrade the application (or at least present a termination message to the user) when this method returns false.

**Building the Java Class**

Compile Listing 16-55 as follows:

```sh
javac Joystick.java
```

**Native Interface Library**

The *native interface library* is a native library that contains JNI function calls and the native code that you want to execute (such as a Win32/Win64 function call that returns the number of supported joystick devices). This native code might serve as an interface between the native interface library and a legacy library.

Before you can code the native interface library, you need to generate a C-based include file that contains C function prototypes corresponding to the native methods. Accomplish this task with the help of the javah (Java header) tool. For example, the following command generates an include file for Listing 16-55; you don’t have to compile the source file before using javah:

```sh
javah Joystick
```

javah analyzes Joystick.java (you cannot specify the .java extension) and generates a file named Joystick.h. This include file is presented in Listing 16-56.

**Listing 16-56. Discovering the C Language Native Function Declaration**

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Joystick */

#ifndef _Included_Joystick
#define _Included_Joystick
#ifdef __cplusplus
extern "C" {
#endif

/* Class: Joystick */
* Method: joyGetNumDevs
* Signature: ()I
*/
```
CHAPTER 16: Focusing on Odds and Ends

Listing 16-56 begins with an #include C language preprocessor directive that includes the contents of a JDK file named jni.h in the source code. This header file declares various structures and JNI functions that a native function uses to cooperate with the Java platform, and it is located in the %JAVA_HOME%\include directory.

The other significant item is JNIEXPORT jint JNICALL Java_Joystick_joyGetNumDevs(JNIEnv *, jclass);. This C function prototype names the native function where the Joystick portion of the name identifies the class in which the corresponding native method is declared. It also describes the parameter list in terms of types only; C doesn’t require names to be specified when declaring a function prototype.

The parameter of JNIEnv * type is a pointer to a C structure that makes it possible to call various JNI functions to perform useful work. No JNI functions are required in this example.

The parameter of jclass type identifies the class of which joyGetNumDevs() is a member. If joyGetNumDevs() was an instance method, this parameter would be of jobject type, which holds a reference to the current object on which the joyGetNumDevs() native method was invoked; think of the jobject parameter as containing this’s value.

Listing 16-57 presents the C code that implements Java_Joystick_joyGetNumDevs(JNIEnv *, jclass).

Listing 16-57. The C Side of the Joystick Hybrid Library

#include <windows.h>
#include "Joystick.h"

JNIEXPORT jint JNIALL Java_Joystick_joyGetNumDevs(JNIEnv *pEnv, jclass clazz)
{
    return joyGetNumDevs();
}

Listing 16-57 first specifies an #include <windows.h> directive, which provides access to Win32’s joyGetNumDevs() multimedia function prototype, and an #include "Joystick.h" directive, which includes jni.h and the Java_Joystick_joyGetNumDevs(JNIEnv *, jclass) function prototype. The native function simply invokes joyGetNumDevs() and returns its value.

Note Much of Listing 16-56 makes sure that the Java_Joystick_joyGetNumDevs(JNIEnv *, jclass) function can be invoked in the context of C++ without C++ name mangling (http://en.wikipedia.org/wiki/Name_mangling) getting in the way.
Building the Native Interface Library

Various C/C++ compilers can be used to create the native interface library. I'm partial to the GNU Compiler Collection (see http://en.wikipedia.org/wiki/GNU_Compiler_Collection), which includes a C compiler.

You can obtain a version of this software for your Unix-oriented platform by pointing your browser to http://gcc.gnu.org/ and following instructions. If your platform is Windows, you will need an environment such as MinGW (http://en.wikipedia.org/wiki/Mingw) to run this compiler. You can obtain a copy of MinGW by pointing your browser to www.mingw.org and following instructions.

Assuming a Windows platform; that Java 7 Update 6 is installed into the c:\progra~1\java\jdk1.7.0_06 home directory (progra~1 is shorthand for Program Files); and that the current directory contains Joystick.java, Joystick.h, and joystick.c (with Listing 16-57's contents); and that you’re using MinGW; execute the following command (spread across two lines here for readability) to compile joystick.c into joystick.o:

```
gcc -c -Ic:\progra~1\java\jdk1.7.0_06\include\ -Ic:\progra~1\java\jdk1.7.0_06\include\win32\ joystick.c
```

The -c option specifies that an object file is to be created and the -I option specifies the include paths for jni.h and jni_md.h—jni_md.h is a machine-dependent file located in the %JAVA_HOME%\include\win32 directory. The end result is an object file named joystick.o.

Execute the following command line to turn this object file into a joystick.dll library:

```
gcc -Wl,--kill-at -shared -o joystick.dll joystick.o -lwinmm
```

The -Wl,--kill-at option specifies no name mangling (which leads to unsatisfied link errors), the -shared option specifies a Windows DLL target, the -o option specifies the output library name, and the -l option specifies a library to be included in the link step. The specified library is winmm, which is an import library needed by Windows to ensure that it can locate the proper DLL containing joyGetNumDevs().

Testing the Hybrid Library

Assuming that you’ve successfully created Joystick.class and the joystick.dll native interface library, you’ll want to test this library’s solitary native function. Listing 16-58 presents the source code to a small application that accomplishes this task.

Listing 16-58. Testing the Joystick Hybrid Library

```java
class JoystickDemo
{
    public static void main(String[] args)
    {
        if (!Joystick.init())
        {
            System.err.println("unable to load joystick library");
            return;
        }
```
System.out.printf("Number of joysticks = %d\n", Joystick.joyGetNumDevs());
}

Compile Listing 16-58 as follows:

cjavac JoystickDemo.java

Run this application as follows:

cjava JoystickDemo

On my Windows 7 platform, I observed the following output:

Number of joysticks = 16

The Windows 7 joystick driver supports a maximum of 16 joystick devices.

**Note** You might run into an unsatisfied link error stating that a 32-bit DLL cannot be loaded onto a 64-bit platform. This error most likely occurs because you’re running a 64-bit version of the virtual machine and attempting to load a 32-bit DLL into this machine. You’ll need to install and run a 32-bit version of the virtual machine to make the example work.


**Focusing on ZIP and JAR**

You might need to develop an application that must create a new ZIP file and store files in that file or extract content from an existing ZIP file. Perhaps you might need to perform either task in the context of a JAR file, which you might think of as a ZIP file with a .jar file extension. This section introduces you to the APIs for working with ZIP and JAR files.

**Focusing on the ZIP API**

The `java.util.zip` package provides classes for working with ZIP files, which are also known as ZIP archives. Each ZIP archive stores files that are typically compressed, and each stored file is known as a ZIP entry. You can use these classes to write ZIP entries to or read ZIP entries from a ZIP archive in the standard ZIP and GZIP (GNU ZIP) file formats; compress and decompress data via the DEFLATE compression algorithm that these formats use; and compute the CRC-32 and Adler-32 checksums of arbitrary input streams.
The ZipEntry class represents a ZIP entry. You must instantiate this class to write new entries to a ZIP archive or read entries from an existing ZIP archive. ZipEntry offers two constructors:

- `ZipEntry(String name)` creates a new ZIP entry with the specified name. This constructor throws `NullPointerException` when `name` is null and `IllegalArgumentException` when the length of the string assigned to `name` exceeds 65,535 bytes.
- `ZipEntry(ZipEntry ze)` creates a new ZIP entry with values taken from existing ZIP entry `ze`.

Additionally, ZipEntry declares several methods including those presented in the following list:

- `String getComment()` returns the entry's comment string or null when there is no comment string. A comment provides user-specific information associated with an entry.
- `long getCompressedSize()` returns the size of the entry's compressed data, or -1 when not specified. The compressed size is the same as the uncompressed size when the entry data is stored without compression.
- `long getCrc()` returns the CRC-32 checksum of the entry's uncompressed data or -1 when the checksum hasn't been specified.
- `int getMethod()` returns the compression method used to compress the entry's data. This value is one of ZipEntry's `DEFLATED` or `STORED` (not compressed) constants or is -1 when the compression method hasn't been specified.
- `String getName()` returns the entry's name.
- `long getSize()` returns the uncompressed size of the entry's data or -1 when the size hasn't been specified.
- `boolean isDirectory()` returns true when the entry describes a directory; otherwise, this method returns false.
- `void setComment(String comment)` sets the entry's comment string to `comment`. A comment string is optional. When specified, the maximum length should be 65,535 bytes; remaining bytes are truncated.
- `void setCompressedSize(long csize)` sets the size (in bytes) of the entry's compressed data to `csize`.
- `void setCrc(long crc)` sets the CRC-32 checksum of the entry's uncompressed data to `crc`. This method throws `IllegalArgumentException` when `crc`'s value is less than 0 or greater than 0xFFFFFFFF.
void setMethod(int method) sets the compression method to method. This method throws IllegalArgumentException when any value other than ZipEntry.DEFLATED (compress data file at a specific level) or ZipEntry.STORED (don't compress) is passed to method.

void setSize(long size) sets the uncompressed size of the entry's data to size. This method throws IllegalArgumentException when size's value is less than 0 or the value is greater than 0xFFFFFFFF when “ZIP64” (http://en.wikipedia.org/wiki/Zip_(file_format)#ZIP64) isn't supported.

You will soon learn how to work with this class.

### Writing Files to a ZIP Archive

Use the ZipOutputStream class to write ZIP entries (compressed as well as uncompressed) to a ZIP archive. ZipOutputStream declares the ZipOutputStream(OutputStream out) constructor for creating a ZIP output stream. Although ZipEntry instances are conceptually written to this stream, it's really the data described by these instances that's written.

**Note** Use the GZIPOutputStream class to create a GZIP archive and write files to this archive in the GZIP format. For brevity, I don't discuss this class.

The following example instantiates ZipOutputStream with an underlying file output stream:

```java
ZipOutputStream zos = new ZipOutputStream(new FileOutputStream("archive.zip"));
```

ZipOutputStream also declares several methods and inherits additional methods from its DeflaterOutputStream superclass. You minimally work with the following methods:

- void close() closes the ZIP output stream along with the underlying output stream.
- void closeEntry() closes the current ZIP entry and positions the stream for writing the next entry.
- void putNextEntry(ZipEntry e) begins writing a new ZIP entry and positions the stream to the start of the entry data. The current entry is closed when still active (that is, when closeEntry() wasn’t invoked on the previous entry).
- void write(byte[] b, int off, int len) writes len bytes starting at offset off from buffer b to the current ZIP entry. This method will block until all the bytes are written.

Each method throws IOException when a generic I/O error has occurred and ZipException (which subclasses IOException) when a ZIP-specific I/O error has occurred.

Listing 16-59 presents a ZipCreate application that shows you how to minimally use ZipOutputStream and ZipEntry to store assorted files in a new ZIP archive.
Listing 16-59. Creating a ZIP Archive and Storing Specified Files in That Archive

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;

class ZipCreate {  
    public static void main(String[] args) throws IOException {  
        if (args.length < 2) {  
            System.err.println("usage: java ZipCreate ZIPfile infile1 "+
                                "infile2 ...");  
            return;
        }
        ZipOutputStream zos = null;
        try {  
            zos = new ZipOutputStream(new FileOutputStream(args[0]));
            byte[] buf = new byte[1024];
            for (String filename: args) {  
                if (filename.equals(args[0])) {  
                    continue;
                }
                FileInputStream fis = null;
                try {  
                    fis = new FileInputStream(filename);  
                    zos.putNextEntry(new ZipEntry(filename));
                    int len;
                    while ((len = fis.read(buf)) > 0) {  
                        zos.write(buf, 0, len);
                    }
                } catch (IOException ioe) {  
                    System.err.println("I/O error: " + ioe.getMessage());
                } finally {  
                    if (fis != null) {  
                        try {  
                            fis.close();
                        } catch (IOException ioe) {  
                            assert false; // shouldn't happen in this context
                        }
                    }
                }
            }
        } finally {  
            if (zos != null) {  
                try {  
                    zos.close();
                } catch (IOException ioe) {  
                    System.err.println("I/O error: " + ioe.getMessage());
                }
            }
        }
    }
}
Listing 16-59 is fairly straightforward. It first validates the number of command-line arguments, which must be at least two: the first argument is always the name of the ZIP file to be created. If successful, this application creates a ZIP output stream with an underlying file output stream to this file and then writes the contents of those files identified by successive command-line arguments to the ZIP output stream.

The only part of this source code that might seem confusing is if (filename.equals(args[0])) continue;. This statement prevents the first command-line argument, which happens to be the name of the ZIP archive, from being added to the archive, which doesn't make sense because of its recursive nature. If this possibility was permitted, a ZipException instance containing a “duplicate entry” message would be thrown.

Compile Listing 16-59 as follows:

javac ZipCreate.java

Run this application via the following command line, which creates a ZIP archive named a.zip and stores file ZipCreate.java in this archive; the application isn’t recursive (it won’t recurse into directories):

java ZipCreate a.zip ZipCreate.java

You shouldn’t observe any output. Instead, you should observe a file named a.zip in the current directory. Furthermore, when you unzip a.zip, you should detect an unarchived ZipCreate.java file.

You cannot store duplicate files in an archive because that makes no sense. For example, you’ll observe an exception message about a duplicate entry when you execute the following command line:

java ZipCreate a.zip ZipCreate.java ZipCreate.java
ZipOutputStream offers more capabilities. For example, you can use its void setLevel(int level) method to set the compression level for successive entries. Specify an integer argument from 0 through 9, where 0 indicates no compression and 9 indicates best compression; better compression slows down performance. (Google reports these limits as -1 and 8.) Alternatively, specify one of the Deflater class’s BEST_COMPRESSION, BEST_SPEED, DEFAULT_COMPRESSION (to which setLevel() defaults), and other constants as an argument.

**Reading Files from a ZIP Archive**

Use the ZipInputStream class to read ZIP entries (compressed as well as uncompressed) from a ZIP archive. ZipInputStream declares the ZipInputStream(InputStream in) constructor for creating a ZIP input stream. Although ZipEntry instances are conceptually read from this stream, it’s really the data described by these instances that’s read.

---

**Note** Use the GZIPInputStream class to open a GZIP archive and read files from this archive in the GZIP format. For brevity, I don’t discuss this class.

---

The following example instantiates ZipInputStream with an underlying file input stream:

```java
ZipInputStream zis = new ZipInputStream(new FileInputStream("archive.zip"));
```

ZipInputStream also declares several methods and inherits additional methods from its InflaterInputStream superclass. You minimally work with the following methods:

- void close() closes the ZIP input stream along with the underlying input stream.
- void closeEntry() closes the current ZIP entry and positions the stream for reading the next entry.
- ZipEntry getNextEntry() reads the next ZIP entry and positions the stream to the start of the entry data. This method returns null when there are no more entries.
- int read(byte[] b, int off, int len) reads a maximum of len bytes from the current ZIP entry into buffer b starting at offset off. This method will block until all of the bytes are read.

Each method throws IOException when a generic I/O error has occurred and (except for close()) ZipException when a ZIP-specific I/O error has occurred. Also, read() throws NullPointerException when b is null and java.lang.IndexOutOfBoundsException when off is negative, len is negative, or len is greater than b.length - off.

Listing 16-60 presents a ZipAccess application that shows you how to minimally use ZipInputStream and ZipEntry to extract assorted files from an existing ZIP archive.
Listing 16-60. Accessing a ZIP Archive and Extracting Specified Files from That Archive

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

public class ZipAccess {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("usage: java ZipAccess zipfile");
            return;
        }
        ZipInputStream zis = null;
        try {
            zis = new ZipInputStream(new FileInputStream(args[0]));
            byte[] buffer = new byte[4096];
            ZipEntry ze;
            while ((ze = zis.getNextEntry()) != null) {
                System.out.println("Extracting: " + ze);
                FileOutputStream fos = null;
                try {
                    fos = new FileOutputStream(ze.getName());
                    int numBytes;
                    while ((numBytes = zis.read(buffer, 0, buffer.length)) != -1)
                        fos.write(buffer, 0, numBytes);
                } catch (IOException ioe) {
                    System.err.println("I/O error: " + ioe.getMessage());
                } finally {
                    if (fos != null)
                        try {
                            fos.close();
                        } catch (IOException ioe) {
                            assert false; // shouldn't happen in this context
                        }
                }
            }
            zis.close();
        } finally {
            if (zis != null)
                try {
                    zis.close();
                } catch (Exception e) {
                    assert false; // shouldn't happen in this context
                }
        }
    }
}
```
Listing 16-60 is fairly straightforward. It first validates the number of command-line arguments, which must be exactly one: the name of the ZIP file to be accessed. Assuming success, it creates a ZIP input stream with an underlying file input stream to this file and then reads the contents of the various files that are stored in this archive, creating these files in the current directory.

Compile Listing 16-60 as follows:

javac ZipAccess.java

Run this application via the following command line, which accesses the previous a.zip archive and extracts file ZipCreate.java from this archive:

java ZipAccess a.zip

You should observe “Extracting: ZipCreate.java” as the single line of output, and you should also note the appearance of a ZipCreate.java file in the current directory.

ZIPFILE VERSUS ZIPINPUTSTREAM

The java.util.zip package contains a ZipFile class that seems to be an alias for ZipInputStream. As with ZipInputStream, you can use ZipFile to read a ZIP file's entries. However, ZipFile has a couple of differences that make it worth considering as an alternative:

- ZipFile allows random access to ZIP entries via its ZipEntry getEntry(String name) method. Given a ZipEntry instance, you can call ZipEntry's InputStream getInputStream(ZipEntry entry) method to obtain an input stream for reading the entry's content. ZipInputStream supports sequential access to ZIP entries.

You might be curious about a ZipFile constructor that declares a mode parameter of type int. The argument passed to mode is ZipFile.OPEN_READ or ZipFile.OPEN_READ | ZipFile.OPEN_DELETE. The latter argument causes the underlying file to be deleted sometime between when it's opened and when it's closed.

This capability was introduced by Java 1.3 to solve a problem related to caching downloaded JAR files in the context of long-running server applications or Remote Method Invocation. The problem is discussed at http://docs.oracle.com/javase/7/docs/technotes/guides/lang/enhancements.html.

---

**Focusing on the JAR API**

The java.util.jar package provides classes for working with JAR files. Because a JAR file is a kind of ZIP file, it isn't surprising that this package provides classes that extend their java.util.zip counterparts. For example, java.util.jar.JarEntry extends java.util.zip.ZipEntry.

The java.util.jar package also provides classes that have no java.util.zip counterparts, for example, Manifest. These classes provide access to JAR-specific capabilities. For example, Manifest lets you work with a JAR file's manifest (explained shortly).

Listing 16-61 presents a MakeRunnableJAR application that shows you how to work with some of the types in the java.util.jar package to create a runnable JAR file.

**Listing 16-61. Creating a Runnable JAR File**

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.jar.Attributes;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.jar.Manifest;

public class MakeRunnableJAR
{
    public static void main(String[] args) throws IOException
    {
        if (args.length < 2)
        {
            System.err.println("usage: java MakeRunnableJAR JARfile " +
                               "classfile1 classfile2 ...");
            return;
        }
        JarOutputStream jos = null;
        try
```
{  Manifest mf = new Manifest();  Attributes attr = mf.getMainAttributes();  attr.put(Attributes.Name.MANIFEST_VERSION, "1.0");  attr.put(Attributes.Name.MAIN_CLASS,    args[1].substring(0, args[1].indexOf('.')));  jos = new JarOutputStream(new FileOutputStream(args[0]), mf);  byte[] buf = new byte[1024];  for (String filename: args)  {    if (filename.equals(args[0]))    continue;    FileInputStream fis = null;    try    {      fis = new FileInputStream(filename);      jos.putNextEntry(new JarEntry(filename));      int len;      while ((len = fis.read(buf)) > 0)      {        jos.write(buf, 0, len);      }    }    catch (IOException ioe)    {      System.err.println("I/O error: " + ioe.getMessage());    }    finally    {      if (fis != null)      {        try        {          fis.close();        }        catch (IOException ioe)        {          assert false; // shouldn't happen in this context        }      }      jos.closeEntry();    }  }  catch (IOException ioe)  {    System.err.println("I/O error: " + ioe.getMessage());  }  finally  {    if (jos != null)    {      try      {        jos.close();      }    }  }
Because Listing 16-61 is very similar to Listing 16-59, albeit with java.util.jar classes replacing their java.util.zip counterparts, I’ll focus on only that part of this application the creates the manifest. However, you first need to understand the concept of a JAR file manifest.

A manifest is a special file named MANIFEST.MF that stores information about the contents of the JAR file. This file is located in the JAR file’s META-INF directory. For example, the manifest would look as follows for an executable hello.jar JAR file containing a Hello application class:

```plaintext
Manifest-Version: 1.0
Main-Class: Hello
```

The first line signifies the version of the manifest and must be present. The second line identifies the application class that is to run when the JAR file is executed. A .class file extension must not be specified. Doing so would suggest that you want to run class class in the Hello package.

**Caution** You must insert an empty line after Main-Class: Hello. Otherwise, you’ll receive a “no main manifest attribute, in hello.jar” error message when trying to run the application.

The key part of Listing 16-61 that sets it apart from Listing 16-59 is the following code fragment:

```java
Manifest mf = new Manifest();
Attributes attr = mf.getMainAttributes();
attr.put(Attributes.Name.MANIFEST_VERSION, "1.0");
attr.put(Attributes.Name.MAIN_CLASS,
    args[1].substring(0, args[1].indexOf('.')));
jos = new JarOutputStream(new FileOutputStream(args[0]), mf);
```

The Manifest class is first instantiated (via its noargument constructor) to describe the soon-to-be-created manifest. Its getMainAttributes() method is then called to return an Attributes instance for accessing existing manifest attributes or creating new manifest attributes (such as Main-Class).

Attributes is essentially a map and provides Object put(Object key, Object value) for storing an attribute name/value pair. The value passed to key must be an Attributes.Name constant such as Attributes.Name.MANIFEST_VERSION or Attributes.Name.MAIN_CLASS.

**Caution** You must store MANIFEST_VERSION; otherwise, you’ll observe a thrown exception at runtime.
Because the `.class` file extension must be specified when specifying the name of a classfile as a command-line argument, expression `args[1].substring(0, args[1].indexOf('.'))` is used to remove this extension. You can specify multiple classfile names as command-line arguments; the first name is stored (without its `.class` extension) in the manifest.

Finally, `JarOutputStream` is instantiated in a similar manner to `ZipOutputStream`. However, the initialized `Manifest` instance is also passed to the constructor as the second argument.

To play with this application, you minimally need a class with a `public static void main(String[] args)` method. For simplicity, consider Listing 16-62.

**Listing 16-62. Saying Hello**

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Listing 16-62 isn’t much of an application, but it is sufficient for our purpose. Compile Listings 16-61 and 16-62 as follows:

```bash
javac MakeRunnableJAR.java
javac Hello.java
```

Now execute the following command line:

```bash
java MakeRunnableJAR hello.jar Hello.class
```

If all goes well, you should observe a `hello.jar` file in the current directory. Execute the following command to run this file:

```bash
java -jar hello.jar
```

Assuming success, you should observe a single line of output consisting of Hello.

---

**EXERCISES**

The following exercises are designed to test your understanding of Chapter 16’s content.

1. Identify the two enhancements to numeric literals introduced by Java 7.
2. What does the diamond operator accomplish?
3. True or false: When multiple exception types are listed in a catch block’s header, the parameter is implicitly regarded as `final`.
4. Identify the statement that Java 7 uses to implement automatic resource management.
5. Define classloader.
6. What is a classloader’s purpose?
7. What classloaders are available when the virtual machine starts running?
8. True or false: The concrete ClassLoader class is the ultimate root class for all classloaders (including extension and system) except for bootstrap.
9. Which classloader is the root classloader?
10. Define current classloader.
11. Define context classloader.
12. Which methods are central to ClassLoader?
13. True or false: Context classloaders can be a source of difficulty.
14. Can you use a classloader to load arbitrary resources (such as images)?
15. What does the Console class provide?
16. How do you obtain the console?
17. True or false: Console’s String readLine() method reads a single line of text (including line-termination characters) from the console’s input stream.
18. Define design pattern.
19. What does the Strategy pattern let you accomplish?
20. What is double brace initialization?
21. Identify the two drawbacks of double brace initialization.
22. Define fluent interface.
23. List five advantages that immutable classes have over mutable classes.
24. List four guidelines for making a class immutable.
25. Define internationalization.
27. Define locale.
28. How does Java represent a locale?
29. Specify an expression for obtaining the Canadian locale.
30. How do you change the default locale that’s made available to the virtual machine?
31. Define resource bundle.
32. True or false: Each resource bundle family shares a common base name.
33. How does an application load its resource bundles?
34. What happens when a resource bundle cannot be found after an exhaustive search?
35. Define property resource bundle.
36. Define list resource bundle.
37. True or false: When a property resource bundle and a list resource bundle have the same complete resource bundle name, the property resource bundle takes precedence over the list resource bundle.

38. What methods does ResourceBundle provide that make it possible to design a server application that clears out all cached resource bundles upon command?

39. Why did Java 6 rework ResourceBundle to depend on a nested Control class?

40. What does the BreakIterator class let you accomplish?

41. What does the Collator class let you accomplish?

42. Define date, time zone, and calendar.

43. How does the Date class represent a date?

44. How do you obtain a calendar for the default locale that uses a specific time zone?

45. True or false: Calendar declares a Date getDate() method that returns a calendar's time representation as a Date instance.

46. Which Format subclass lets you obtain formatters that format numbers as currencies, integers, numbers with decimal points, and percentages (and also to parse such values)?

47. How would you obtain a date formatter to format the time portion of a date in a particular style for the default locale?

48. What is the difference between a simple message and a compound message?

49. How do you format a simple message and a compound message?

50. Which class does Java provide to format simple and compound messages?

51. What class should you use to format a compound message that contains singular and plural words?

52. True or false: The Format class declares parseObject() methods for parsing strings into objects.

53. Identify the package associated with Java's Logging API.

54. Which class is the entry-point into the Logging API?

55. Identify the standard set of log level constants provided by the Level class?

56. True or false: Loggers default to outputting log records for all log levels.

57. How do you obtain a logger?

58. How do you obtain a logger's nearest parent logger?

59. Identify the four categories of message-logging methods.

60. At what log level do the entering() and exiting() methods log messages?

61. How do you change a logger's log level?

62. What method does the Filter interface provide for further filtering log records regardless of their log levels?

63. How do you obtain the name of the method that's the source of a log record?
64. How does the logging framework send a log record to its ultimate destination (such as the console)?
65. How do you obtain a logger’s registered handlers?
66. True or false: FileHandler’s default formatter is an instance of SimpleFormatter.
67. What does the LogManager class accomplish?
68. Explain the need for the ErrorManager class.
69. What does the Preferences API let you accomplish?
70. How does the Preferences API manage preferences?
71. What is the difference between the system preference tree and the user preference tree?
72. Identify the package assigned to the Preferences API and list its members.
73. Which one of the Preferences API’s types is the entry point?
74. How do you obtain the root node of the system preference tree?
75. What does the Runtime class accomplish, and how do you obtain an instance of this class?
76. What does Runtime provide for executing other applications?
77. Define Java Native Interface.
78. What is a hybrid library?
79. What does the JNI do when you try to load a nonexistent library?
80. Identify the package for working with ZIP archives.
81. Each stored file in a ZIP archive is known as what?
82. True or false: The name of a stored file in a ZIP archive cannot exceed 65,536 bytes.
83. What does ZipOutputStream accomplish?
84. What does ZipInputStream accomplish?
85. Which class appears to be an alias for ZipInputStream?
86. Identify the package for working with JAR files.
87. Define manifest.
88. True or false: When creating a manifest for a JAR output stream, you must store MANIFEST_VERSION; otherwise, you’ll observe a thrown exception at runtime.
89. Create a SpanishCollation application that outputs Spanish words ñango (weak), llamado (called), lunes (Monday), champán (champagne), clamor (outcry), cerca (near), nombre (name), and chiste (joke) according to this language’s current collation rules followed by its traditional collation rules. According to the current collation rules, the output order is as follows: cerca, champán, chiste, clamor, llamado, lunes, nombre, and ñango. According to the traditional collation rules, the output order is as follows: cerca, clamor, champán, chiste, lunes, llamado, nombre, and ñango. Use the RuleBasedCollator class to specify the rules for traditional collation. Also, construct your Locale object using only the es (Spanish) language code.
Note The Spanish alphabet consists of 29 letters: a, b, c, ch, d, e, f, g, h, i, j, k, l, ll, m, n, ñ, o, p, q, r, s, t, u, v, w, x, y, z. (Vowels are often written with accents, as in tablón [plank or board], and u is sometimes topped with a dieresis or umlaut, as in vergüenza [bashfulness]. However, vowels with these diacritical marks are not considered separate letters.) Before April 1994’s voting at the X Congress of the Association of Spanish Language Academies, ch was collated after c, and ll was collated after l. Because this congress adopted the standard Latin alphabet collation rules, ch is now considered a sequence of two distinct characters and dictionaries now place words starting with ch between words starting with cg and ci. Similarly, ll is now considered a sequence of two characters.

90. Create a ZipList application that's similar to ZipAccess but only outputs information about the archive; it doesn’t extract file contents as well. Information to be output is the name of the entry, the compressed and uncompressed sizes, and the last modification time. Use the Date class to convert the last modification time to a human-readable string.

Summary

Apache Harmony is the basis for the core of Android’s standard class library, which is why Android doesn’t support Java language features more recent than Java 5 and APIs more recent than Java 6. However, it’s possible to add this support.

Java hasn’t stopped evolving, and version 7 introduced several new language features that will be helpful to Android app developers. These features range from the small (adding underscores to integer literals, for example) to the more significant (such as automatic resource management).

The virtual machine relies on classloaders to dynamically load compiled classes and other reference types (classes, for short) from classfiles, JAR files, URLs, and other sources into memory. Classloaders insulate the virtual machine from filesystems, networks, and so on.

When the virtual machine starts running, bootstrap, extension, and system classloaders are available. Also, the standard class library provides the abstract ClassLoader class as the ultimate root class for all classloaders (including extension and system) except for bootstrap.

ClassLoader is subclassed by the concrete SecureClassLoader class, which takes security information into account. SecureClassLoader is subclassed by the concrete URLClassLoader class, which lets you load classes and resources from a search path of URLs referring to JAR files and directories.

Starting with Java 1.2, classloaders have a hierarchical relationship in which each classloader except for bootstrap has a parent classloader. The bootstrap classloader is the root classloader in much the same way as Object is the root reference type.

Java also recognizes current and context classloaders. The current classloader is the classloader that loads the class to which the currently executing method belongs. The context classloader (introduced by Java 1.2) is the classloader associated with the current thread.
Central to ClassLoader are its `Class<?> loadClass(String name)` and protected `Class<?> loadClass(String name, boolean resolve)` methods, which try to load the class with the specified name. They throw `ClassNotFoundException` when the class cannot be found or return a `Class` object representing the loaded class.

Classloaders are typically used to load classes, but they can also load arbitrary resources (such as images) via `ClassLoader` methods such as `InputStream getResourceAsStream(String name)`. Although you could call these methods directly, it's common practice to work with `Class`'s URL `getResources(String name)` and `InputStream getResourceAsStream(String name)` methods instead.

Java 6 introduced a `Console` class that facilitates the development of console-based applications. For example, `Console` provides a `readPassword()` method for prompting the user to enter a password without echoing the password to the screen.

Designing significant applications is often difficult and prone to error. Over the years, developers have encountered various design problems and have devised clever solutions. These problems and their solutions have been catalogued to help other developers detect them and avoid reinventing solutions. These catalogued entities are known as design patterns.

A design pattern is a catalogued problem/solution entity. It consists of a name describing the pattern and a vocabulary for discussing it with other developers, a clear statement of the problem that the design pattern solves, a solution in terms of the classes and objects and their relationships and other factors that solve the problem, and the consequences of using the pattern.

Double brace initialization is a special syntax for initializing a newly created object in a compact manner, for example, `new Office() {{addEmployee(new Employee("John Doe"));}};`. Because this syntax sugar is based on an anonymous class, one drawback with this technique is a bloated number of classfiles. A second drawback: you cannot use Java 7’s diamond operator.

Method chaining is used to construct fluent interfaces, which are implementations of object-oriented APIs that provide for more readable code. A fluent interface is normally implemented via method chaining to relay the instruction context of a subsequent method call. The context is defined through the return value of a called method, the context is self-referential in that the new context is equivalent to the last context, and the context is terminated through the return of a void context.

The `final` keyword plays a large role in the creation of immutable classes, which are classes whose instances cannot be modified. Immutable classes offer several advantages, which is why you might consider designing most of your classes to be immutable. Most importantly, objects created from immutable classes are thread-safe and there are no synchronization issues.

There are several guidelines that must be followed to ensure that a class is immutable: don’t include setter or other mutator methods in the class design, prevent methods from being overridden (typically by marking the class `final`), declare all fields `final`, and prevent the class from exposing any mutable state.

Internationalization is the process of creating an application that automatically adapts to its current user’s culture (without recompilation) so that the user can read text in the user’s language and otherwise interact with the application without observing cultural issues. Related to internationalization is localization, which is the adaptation of internationalized software to support a new culture by adding culture-specific elements (such as text strings that have been translated to the culture).
The Locale class is the centerpiece of the various Internationalization APIs. Instances of this class represent locales, which are geographical, political, or cultural regions.

An internationalized application contains no hard-coded text or other locale-specific elements (such as a specific currency format). Instead, each supported locale’s version of these elements is stored outside of the application. Java is responsible for storing each locale’s version of certain elements, such as currency formats. In contrast, it’s your responsibility to store each supported locale’s version of other elements, such as text, audio clips, and locale-sensitive images. These elements are typically stored in resource bundles, which are containers that store locale-specific elements.

Java distinguishes between property resource bundles, which are backed by text-based properties files, and list resource bundles, which are Java classes that extend ListResourceBundle, and which can contain binary data.

Internationalized text-processing applications need to detect logical boundaries within the text they’re manipulating. For example, a word processor needs to detect these boundaries when highlighting a character, selecting a word to cut to the clipboard, moving the caret (text insertion point indicator) to the start of the next sentence, and wrapping a word at the end of a line. Java provides the BreakIterator API with its abstract BreakIterator entry-point class to detect text boundaries.

Applications perform string comparisons while sorting text. When an application targets English-oriented users, String’s compareTo() method is probably sufficient for comparing strings. However, this method’s binary comparison of each string’s Unicode characters isn’t reliable for languages where the relative order of their characters doesn’t correspond to the Unicode values of these characters. French is one example. Java provides the Collator API with its abstract Collator entry-point class for making reliable comparisons.

Internationalized applications must properly handle dates, time zones, and calendars. A date is a recorded temporal moment, a time zone is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a calendar is a system of organizing the passage of time.

Java 1.0 introduced the Date class as its first attempt to describe calendars. However, Date was not amenable to internationalization because of its English-oriented nature and because of its inability to represent dates prior to midnight January 1, 1970 GMT, which is known as the Unix epoch (the date when Unix began to be used).

Date’s toString() method reveals that a time zone is part of a date. Java provides the abstract TimeZone entry-point class for obtaining instances of TimeZone subclasses.

Java 1.1 introduced the Calendar API with its abstract Calendar entry-point class as a replacement for Date. Calendar is intended to represent any kind of calendar. However, time constraints meant that only the Gregorian calendar could be implemented (via the concrete GregorianCalendar subclass) for version 1.1.

Internationalized applications don’t present unformatted numbers (including currencies and percentages), dates, and messages to the user. These items must be formatted according to the user’s locale so that they appear meaningful. To help with formatting, Java provides the abstract Format class and various subclasses.
The abstract NumberFormat entry-point class declares class methods to return formatters that format numbers as currencies, integers, numbers with decimal points, and percentages (and also to parse such values). The abstract DateFormat entry-point class provides access to formatters that format Date instances as dates or time values (and also to parse such values). The concrete MessageFormat class lets you format simple and compound messages. For a simple message, you obtain its text from a resource bundle and then display this text to the user. For a compound message, you obtain a pattern (template) for the message from a property resource bundle, pass this pattern and the variable data to a message formatter to create a simple message, and display this message’s text.

In Chapter 5, I presented a simple logging framework to demonstrate packages. Creating your own logging framework is typically a waste of time, and you should use the standard java.util.logging package instead. This package implements Java's Logging API, which was introduced in Java 1.4.

The java.util.logging package consists of 2 interfaces and 15 classes. This package’s key types include Logger (the entry-point into the Logging API; it lets you log messages to various destinations such as files or the console), LogRecord (a description of a message as a record; used to pass messages between the logging framework and individual Handlers), Handler (a receiver of messages from Logger and a publisher of these messages to a console, a file, a network logging service, and so on), Level (a set of standard log levels that can be used to control logging output), Filter (an interface that provides fine-grain control over what is logged, beyond the control provided by log levels), and Formatter (an infrastructure for formatting LogRecords into strings).

Significant applications have preferences, which are configuration items. The Preferences API lets you store preferences in a hierarchical manner so that you can avoid name collisions. Because this API is backend-neutral, it doesn’t matter where the preferences are stored (a file, a database, or [on Windows platforms] the registry); you don’t have to hardcode file names and locations. Also, there are no text files that can be modified, and Preferences can be used on diskless platforms.

This API uses trees of nodes to manage preferences. These nodes are the analogue of a hierarchical filesystem’s directories. Also, preference name/value pairs stored under these nodes are the analogues of a directory’s files. You navigate these trees in a similar manner to navigating a filesystem: specify an absolute path starting from the root node (/) to the node of interest, such as /window/location and /window/size.

There are two kinds of trees: system and user. All users share the system preference tree, whereas the user preference tree is specific to a single user, which is generally the person who logged into the underlying platform. (The precise description of “user” and “system” varies from implementation to implementation of the Preferences API.)

Although the Preferences API’s java.util.prefs package contains three interfaces (NodeChangeListener, PreferencesChangeListener, and PreferencesFactory), four regular classes (AbstractPreferences, NodeChangeEvent, PreferenceChangeEvent, and Preferences), and two exception classes (BackingStoreException and InvalidPreferencesFormatException), you mostly work with the Preferences class.

The Runtime class provides Java applications with access to their runtime environment. An instance of this class is obtained by invoking its Runtime.getRuntime() class method.
Runtime declares several methods that are also declared in System. For example, Runtime declares a void gc() method. Behind the scenes, System defers to its Runtime counterpart by first obtaining the Runtime instance and then invoking this method via that instance. For example, System’s static void gc() method executes Runtime.getRuntime().gc();.

Some of Runtime’s methods are dedicated to executing other applications. For example, Process exec(String program) executes the program named program in a separate native process. The new process inherits the environment of the method’s caller, and a Process object is returned to allow communication with the new process. IOException is thrown when an I/O error occurs.

The Java Native Interface is a native programming interface that lets Java code running in a virtual machine interoperate with native libraries written in other languages (such as C, C++, or even assembly language). The JNI is a bridge between the virtual machine and the underlying platform.

The java.util.zip package provides classes for working with ZIP files, which are also known as ZIP archives. Each ZIP archive stores files that are typically compressed, and each stored file is known as a ZIP entry. You can use these classes to write ZIP entries to or read ZIP entries from a ZIP archive in the standard ZIP and GZIP (GNU ZIP) file formats, compress and decompress data via the DEFLATE compression algorithm that these formats use, and compute the CRC-32 and Adler-32 checksums of arbitrary input streams.

The java.util.jar package provides classes for working with JAR files. Because a JAR file is a kind of ZIP file, it isn’t surprising that this package provides classes that extend their java.util.zip counterparts. For example, JarEntry extends ZipEntry.

The java.util.jar package also provides classes that have no java.util.zip counterparts, such as Manifest. These classes provide access to JAR-specific capabilities. For example, Manifest lets you work with a JAR file’s manifest.

Now that you’ve reached the end of this chapter, check out Appendixes A and B, which offer solutions to all exercises in Chapters 1 through 16 and introduce you to a card game application.
Solutions to Exercises

Each of Chapters 1 through 16 closes with an “Exercises” section that tests your understanding of the chapter’s material. Solutions to these exercises are presented in this appendix.

Chapter 1: Getting Started with Java

1. Java is a language and a platform. The language is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers. The platform consists of a virtual machine and an associated execution environment.

2. A virtual machine is a software-based processor that presents its own instruction set.

3. The purpose of the Java compiler is to translate source code into instructions (and associated data) that are executed by the virtual machine.

4. The answer is true: a classfile's instructions are commonly referred to as bytecode.

5. When the JVM’s interpreter learns that a sequence of bytecode instructions is being executed repeatedly, it informs the JVM’s just-in-time (JIT) compiler to compile these instructions into native code.

6. The Java platform promotes portability by providing an abstraction over the underlying platform. As a result, the same bytecode runs unchanged on Windows-based, Linux-based, Mac OS X–based, and other platforms.

7. The Java platform promotes security by doing its best to provide a secure environment in which code executes. It accomplishes this task in part by using a bytecode verifier to make sure that the classfile’s bytecode is valid.
8. The answer is false: Java SE is the platform for developing applications and applets.

9. The JRE implements the Java SE platform and makes it possible to run Java programs.

10. The difference between the public and private JREs is that the public JRE exists apart from the JDK, whereas the private JRE is a component of the JDK that makes it possible to run Java programs independently of whether or not the public JRE is installed.

11. The JDK is a software development kit that provides tools (including a compiler) for developing Java programs. It also provides a private JRE for running these programs.

12. The JDK’s javac tool is used to compile Java source code.

13. The JDK’s java tool is used to run Java applications.

14. Standard I/O is a mechanism consisting of Standard Input, Standard Output, and Standard Error that makes it possible to read text from different sources (keyboard or file), write nonerror text to different destinations (screen or file), and write error text to different destinations (screen or file).

15. You specify the main() method’s header as public static void main(String[] args).

16. An IDE is a development framework consisting of a project manager for managing a project’s files, a text editor for entering and editing source code, a debugger for locating bugs, and other features. The IDE that Google supports for developing Android apps is Eclipse.

17. Android is Google’s software stack for mobile devices. This stack consists of apps (such as Browser and Contacts), a virtual machine in which apps run, middleware (software that sits on top of the operating system and provides various services to the virtual machine and its apps), and a Linux-based operating system.

18. The API level associated with Android 4.4 is 19.

19. The DEX format is Android’s executable format for apps. DEX is optimized for a minimal memory footprint.

20. Android uses the dx tool to transform compiled Java classfiles into the DEX format.
Chapter 2: Learning Language Fundamentals

1. Unicode is a computing industry standard for consistently encoding, representing, and handling text that’s expressed in most of the world’s writing systems.

2. A comment is a language feature for embedding documentation in source code.

3. The three kinds of comments that Java supports are single-line, multiline, and Javadoc.

4. An identifier is a language feature that consists of letters (A-Z, a-z, or equivalent uppercase/lowercase letters in other human alphabets), digits (0-9 or equivalent digits in other human alphabets), connecting punctuation characters (such as the underscore), and currency symbols (such as the dollar sign, $). This name must begin with a letter, a currency symbol, or a connecting punctuation character; and its length cannot exceed the line in which it appears.

5. The answer is false: Java is a case-sensitive language.

6. A type is a language feature that identifies a set of values (and their representation in memory) and a set of operations that transform these values into other values of that set.

7. A primitive-type is a type that’s defined by the language and whose values are not objects.

8. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double precision floating-point primitive-types.

9. A user-defined type is a type that’s defined by the developer using a class, an interface, an enum, or an annotation type and whose values are objects.

10. An array type is a special reference type that signifies an array, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as elements.

11. A variable is a named memory location that stores some type of value.

12. An expression is a combination of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression’s type.

13. The two expression categories are simple expression and compound expression.

14. A literal is a value specified verbatim.
15. String literal "The quick brown fox \jumps\ over the lazy dog." is illegal because, unlike ", \j and \ (a backslash followed by a space character) are not valid escape sequences. To make this string literal legal, you must escape these backslashes, as in "The quick brown fox \\jumps\\ over the lazy dog."

16. An operator is a sequence of instructions symbolically represented in source code.

17. The difference between a prefix operator and a postfix operator is that a prefix operator precedes its operand and a postfix operator trails its operand.

18. The purpose of the cast operator is to convert from one type to another type. For example, you can use this operator to convert from floating-point type to 32-bit integer type.

19. Precedence refers to an operator’s level of importance.

20. The answer is true: most of Java’s operators are left-to-right associative.

21. A statement is a language feature that assigns a value to a variable, controls a program’s flow by making a decision and/or repeatedly executing another statement, or performs another task.

22. The difference between the while and do-while statements is that the while statement evaluates its Boolean expression at the top of the loop, whereas the do-while statement evaluates its Boolean expression at the bottom of the loop. As a result, while executes zero or more times, whereas do-while executes one or more times.

23. The difference between the break and continue statements is that break transfers execution to the first statement following a switch statement or a loop, whereas continue skips the remainder of the current loop iteration, reevaluates the loop’s Boolean expression, and performs another iteration (when true) or terminates the loop (when false).

24. Listing A-1 presents the Compass application that was called for in Chapter 2.

Listing A-1. Finding a Direction in Which to Travel

```java
public class Compass {
    public static void main(String[] args) {
        int direction = 1;
        switch (direction) {
            case 0: System.out.println("You are travelling north."); break;
            case 1: System.out.println("You are travelling east."); break;
            case 2: System.out.println("You are travelling south."); break;
        }
    }
}
```
```java
case 3: System.out.println("You are travelling west."); break;
default: System.out.println("You are lost.");
}
}
}

25. Listing A-2 presents the Triangle application that was called for in Chapter 2.

**Listing A-2. Printing a Triangle of Asterisks**

```java
public class Triangle {
    public static void main(String[] args) {
        for (int row = 1; row < 20; row += 2) {
            for (int col = 0; col < 19 - row / 2; col++)
                System.out.print(" ");
            for (int col = 0; col < row; col++)
                System.out.print("*");
            System.out.print(\n');
        }
    }
}
```

26. Listing A-3 presents the first PromptForC application that was called for in Chapter 2.

**Listing A-3. Looping Until the User Enters C or c (Version 1)**

```java
public class PromptForC {
    public static void main(String[] args) throws java.io.IOException {
        int ch = 0;
        while (ch != 'C' && ch != 'c') {
            System.out.println("Press C or c to continue.");
            ch = System.in.read();
        }
    }
}
```

Listing A-4 presents the second PromptForC application that was called for in Chapter 2.

**Listing A-4. Looping Until the User Enters C or c (Version 2)**

```java
public class PromptForC {
    public static void main(String[] args) throws java.io.IOException {
        int ch;
```

www.it-ebooks.info
do
{
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
}
while (ch != 'C' && ch != 'c');

Chapter 3: Discovering Classes and Objects

1. A class is a container for housing an application and is also a template for manufacturing objects.

2. You declare a class by minimally specifying reserved word class followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code), followed by a body. The body starts with an open brace character (\{) and ends with a close brace (\}). Sandwiched between these delimiters are various kinds of member declarations.

3. The answer is false: you can declare only one public class in a source file.

4. An object is an instance of a class.

5. You obtain an object by using the new operator to allocate memory to store a class instance and a constructor to initialize this instance.

6. A constructor is a block of code for constructing an object by initializing it in some manner.

7. The answer is true: Java creates a default no-argument constructor when a class declares no constructors.

8. A parameter list is a round bracket-delimited and comma-separated list of zero or more parameter declarations. A parameter is a constructor or method variable that receives an expression value passed to the constructor or method when it’s called.

9. An argument list is a round bracket-delimited and comma-separated list of zero or more expressions. An argument is one of these expressions whose value is passed to the corresponding parameter when a constructor or method is called.

10. The answer is false: you invoke another constructor by specifying this followed by an argument list.

11. Arity is the number of arguments passed to a constructor or method or the number of operator operands.
12. A local variable is a variable that's declared in a constructor or method and isn't a member of the constructor or method parameter list.

13. Lifetime is a property of a variable that determines how long the variable exists. For example, parameters come into existence when a constructor or method is called and are destroyed when the constructor or method finishes. Similarly, an instance field comes into existence when an object is created and is destroyed when the object is garbage collected.

14. Scope is a property of a variable that determines how accessible the variable is to code. For example, a parameter can be accessed only by the code within the constructor or method in which the parameter is declared.

15. Encapsulation refers to the merging of state and behaviors into a single source code entity. Instead of separating state and behaviors, which is done in structured programs, state and behaviors are combined into classes and objects, which are the focus of object-based programs. For example, where a structured program makes you think in terms of separate balance state and deposit/withdraw behaviors, an object-based program makes you think in terms of bank accounts, which unite balance state with deposit/withdraw behaviors through encapsulation.

16. A field is a variable declared within a class body.

17. The difference between an instance field and a class field is that an instance field describes some attribute of the real-world entity that an object is modeling and is unique to each object, and a class field identifies some data item that's shared by all objects.

18. A blank final is a read-only instance field. It differs from a true constant in that there are multiple copies of blank finals (one per object) and only one true constant (one per class).

19. You prevent a field from being shadowed by changing the name of a same-named local variable or parameter, or by qualifying the local variable's name or parameter's name with this or the class name followed by the member access operator.

20. A method is a named block of code declared within a class body.

21. The difference between an instance method and a class method is that an instance method describes some behavior of the real-world entity that an object is modeling and can access a specific object's state, whereas a class method identifies some behavior that's common to all objects and cannot access a specific object's state.

22. Recursion is the act of a method invoking itself.

23. You overload a method by introducing a method with the same name as an existing method but with a different parameter list into the same class.
24. A class initializer is a static-prefixed block that's introduced into a class body. An instance initializer is a block that's introduced into a class body as opposed to being introduced as the body of a method or a constructor.

25. A garbage collector is code that runs in the background and occasionally checks for unreferenced objects.

26. An object graph is a hierarchy of all of the objects currently stored in the heap.

27. The answer is false: String[] letters = new String[2] { "A", "B" }; is incorrect syntax. Remove the 2 from between the square brackets to make it correct.

28. A ragged array is a two-dimensional array in which each row can have a different number of columns.

29. Listing A-5 presents the Image application that was called for in Chapter 3.

Listing A-5. Testing the Image Class

```java
public class Image {
    Image()
    {
        System.out.println("Image() called");
    }

    Image(String filename)
    {
        this(filename, null);
        System.out.println("Image(String filename) called");
    }

    Image(String filename, String imageType)
    {
        System.out.println("Image(String filename, String imageType) called");
        if (filename != null)
        {
            System.out.println("reading " + filename);
            if (imageType != null)
                System.out.println("interpreting " + filename + " as storing a " + imageType + " image");
        }
        // Perform other initialization here.
    }

    public static void main(String[] args)
    {
        Image image = new Image();
        System.out.println();
        image = new Image("image.png");
    }
}
```

www.it-ebooks.info
30. Listing A-6 presents the Conversions application that was called for in Chapter 3.

Listing A-6. Converting Between Degrees Fahrenheit and Degrees Celsius

```java
public class Conversions {
    static double c2f(double degrees) {
        return degrees * 9.0 / 5.0 + 32;
    }

    static double f2c(double degrees) {
        return (degrees - 32) * 5.0 / 9.0;
    }

    public static void main(String[] args) {
        System.out.println("Fahrenheit equivalent of 100 degrees Celsius is " +
                          Conversions.c2f(100));
        System.out.println("Celsius equivalent of 98.6 degrees Fahrenheit is " +
                          Conversions.f2c(98.6));
        System.out.println("Celsius equivalent of 32 degrees Fahrenheit is " +
                          f2c(32));
    }
}
```

31. Listing A-7 presents the Utilities application that was called for in Chapter 3.

Listing A-7. Calculating Factorials and Summing a Variable Number of Double Precision Floating-Point Values

```java
public class Utilities {
    static int factorial1(int n) {
        int product = 1;
        for (int i = 2; i <= n; i++)
            product *= i;
        return product;
    }

    static int factorial2(int n) {
        if (n == 0 || n == 1) // base problem
            return 1;
        return factorial1(n-1) + factorial2(n-1);
    }
}
```
else
    return n * factorial2(n - 1);
}

static double sum(double... values)
{
    int total = 0;
    for (int i = 0; i < values.length; i++)
        total += values[i];
    return total;
}

public static void main(String[] args)
{
    System.out.println(factorial1(4));
    System.out.println(factorial2(4));
    System.out.println(factorial2(0));
    System.out.println(factorial2(1));
    System.out.println(sum(10.0, 20.0));
    System.out.println(sum(30.0, 40.0, 50.0));
}

32. Listing A-8 presents the GCD application that was called for in Chapter 3.

**Listing A-8. Recursively Calculating the Greatest Common Divisor**

```java
class GCD
{
    public static int gcd(int a, int b)
    {
        // The greatest common divisor is the largest positive integer that
        // divides evenly into two positive integers a and b. For example,
        // GCD(12, 18) is 6.
        if (b == 0) // Base problem
            return a;
        else
            return gcd(b, a % b);
    }

    public static void main(String[] args)
    {
        System.out.println(gcd(12, 18));
    }
}
```

www.it-ebooks.info
33. Listing A-9 presents the Book application that was called for in Chapter 3.


```java
public class Book {
    private String name;
    private String author;
    private String isbn;

    public Book(String name, String author, String isbn) {
        this.name = name;
        this.author = author;
        this.isbn = isbn;
    }

    public String getName() {
        return name;
    }

    public String getAuthor() {
        return author;
    }

    public String getISBN() {
        return isbn;
    }

    public static void main(String[] args) {
        Book[] books = new Book[] {
            new Book("Jane Eyre", "Charlotte Brontë", "0895772000"),
            new Book("A Kick in the Seat of the Pants", "Roger von Oech", "0060155280"),
            new Book("The Prince and the Pilgrim", "Mary Stewart", "0340649925")
        };

        for (int i = 0; i < books.length; i++)
            System.out.println(books[i].getName() + " - " + books[i].getAuthor() + " - " + books[i].getISBN());
    }
}
```
Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces

1. Implementation inheritance is inheritance through class extension.

2. Java supports implementation inheritance by providing reserved word extends.

3. A subclass can have only one superclass because Java doesn’t support multiple implementation inheritance.

4. You prevent a class from being subclassed by declaring the class final.

5. The answer is false: the super() call can only appear in a constructor.

6. If a superclass declares a constructor with one or more parameters, and if a subclass constructor doesn’t use super() to call that constructor, the compiler reports an error because the subclass constructor attempts to call a nonexistent noargument constructor in the superclass. (When a class doesn’t declare any constructors, the compiler creates a constructor with no parameters [a noargument constructor] for that class. Therefore, if the superclass didn’t declare any constructors, a noargument constructor would be created for the superclass. Continuing, if the subclass constructor didn’t use super() to call the superclass constructor, the compiler would insert the call and there would be no error.)

7. An immutable class is a class whose instances cannot be modified.

8. The answer is false: a class cannot inherit constructors.

9. Overriding a method means replacing an inherited method with another method that provides the same signature and the same return type but provides a new implementation.

10. To call a superclass method from its overriding subclass method, prefix the superclass method name with reserved word super and the member access operator in the method call.

11. You prevent a method from being overridden by declaring the method final.

12. You cannot make an overriding subclass method less accessible than the superclass method it is overriding because subtype polymorphism would not work properly if subclass methods could be made less accessible. Suppose you upcast a subclass instance to superclass type by assigning the instance’s reference to a variable of superclass type. Now suppose you specify a superclass method call on the variable. If this method is overridden by the subclass, the subclass version of the method is called. However, if access to the subclass’s overriding method’s access could be made private, calling this method would break encapsulation; private methods cannot be called directly from outside of their class.
13. You tell the compiler that a method overrides another method by prefixing the overriding method's header with the `@Override` annotation.

14. Java doesn't support multiple implementation inheritance because this form of inheritance can lead to ambiguities.

15. The name of Java's ultimate superclass is `Object`. This class is located in the `java.lang` package.

16. The purpose of the `clone()` method is to duplicate an object without calling a constructor.

17. `Object`'s `clone()` method throws `CloneNotSupportedException` when the class whose instance is to be shallowly cloned doesn't implement the `Cloneable` interface.

18. The difference between shallow copying and deep copying is that shallow copying copies each primitive or reference field's value to its counterpart in the clone, whereas deep copying creates, for each reference field, a new object and assigns its reference to the field. This deep copying process continues recursively for these newly created objects.

19. The `==` operator cannot be used to determine if two objects are logically equivalent because this operator only compares object references and not the contents of these objects.

20. `Object`'s `equals()` method compares the current object's this reference to the reference passed as an argument to this method. (When I refer to `Object`'s `equals()` method, I am referring to the `equals()` method in the `Object` class.)

21. Expression "abc" == "a" + "bc" returns true. It does so because the `String` class contains special support that allows literal strings and string-valued constant expressions to be compared via `==`.

22. You can optimize a time-consuming `equals()` method by first using `==` to determine if this method's reference argument identifies the current object (which is represented in source code via reserved word `this`).

23. The purpose of the `finalize()` method is to provide a safety net for calling an object's cleanup method in case that method isn't called.

24. You shouldn't rely on `finalize()` for closing open files because file descriptors are a limited resource and an application might not be able to open additional files until `finalize()` is called, and this method might be called infrequently (or perhaps not at all).

25. A hash code is a small value that results from applying a mathematical function to a potentially large amount of data.
26. The answer is true: you should override the `hashCode()` method whenever you override the `equals()` method.

27. Object’s `toString()` method returns a string representation of the current object that consists of the object’s class name, followed by the `@` symbol, followed by a hexadecimal representation of the object’s hash code. (When I refer to Object’s `toString()` method, I’m referring to the `toString()` method in the `Object` class.)

28. You should override `toString()` to provide a concise but meaningful description of the object to facilitate debugging via `System.out.println()` method calls. It’s more informative for `toString()` to reveal object state than to reveal a class name, followed by the `@` symbol, followed by a hexadecimal representation of the object’s hash code.

29. Composition is a way to reuse code by composing classes out of other classes based on a “has-a” relationship between them.

30. The answer is false: composition is used to describe “has-a” relationships and implementation inheritance is used to describe “is-a” relationships.

31. The fundamental problem of implementation inheritance is that it breaks encapsulation. You fix this problem by ensuring that you have control over the superclass as well as its subclasses, by ensuring that the superclass is designed and documented for extension, or by using a wrapper class in lieu of a subclass when you would otherwise extend the superclass.

32. Subtype polymorphism is a kind of polymorphism where a subtype instance appears in a supertype context, and executing a supertype operation on the subtype instance results in the subtype’s version of that operation executing.

33. Subtype polymorphism is accomplished by upcasting the subtype instance to its supertype; by assigning the instance’s reference to a variable of that type; and, via this variable, calling a superclass method that’s been overridden in the subclass.

34. You would use abstract classes and abstract methods to describe generic concepts (such as shape, animal, or vehicle) and generic operations (such as drawing a generic shape). Abstract classes cannot be instantiated and abstract methods cannot be called because they have no code bodies.

35. An abstract class can contain concrete methods.

36. The purpose of downcasting is to access subtype features. For example, you would downcast a `Point` variable that contains a `Circle` instance reference to the `Circle` type so that you can call `Circle`’s `getRadius()` method on the instance.
37. Two forms of RTTI are the virtual machine verifying that a cast is legal and using the `instanceof` operator to determine whether or not an instance is a member of a type.

38. A covariant return type is a method return type that, in the superclass’s method declaration, is the supertype of the return type in the subclass’s overriding method declaration.

39. You formally declare an interface by specifying at least reserved word `interface`, followed by a name, followed by a brace-delimited body of constants and/or method headers.

40. The answer is true: you can precede an interface declaration with the `abstract` reserved word. However, doing so is redundant.

41. A marker interface is an interface that declares no members.

42. Interface inheritance is inheritance through interface implementation or interface extension.

43. You implement an interface by appending an implements clause, consisting of reserved word `implements` followed by the interface’s name, to a class header and by overriding the interface’s method headers in the class.

44. You might encounter one or more name collisions when you implement multiple interfaces.

45. You form a hierarchy of interfaces by appending reserved word `extends` followed by an interface name to an interface header.

46. Java’s interfaces feature is so important because it gives developers the utmost flexibility in designing their applications.

47. Interfaces and abstract classes describe abstract types.

48. Interfaces and abstract classes differ in that interfaces can only declare abstract methods and constants and can be implemented by any class in any class hierarchy. In contrast, abstract classes can declare constants and nonconstant fields; can declare abstract and concrete methods; and can only appear in the upper levels of class hierarchies, where they’re used to describe abstract concepts and behaviors.

49. Listings A–10 through A–16 declare the `Animal`, `Bird`, `Fish`, `AmericanRobin`, `DomesticCanary`, `RainbowTrout`, and `SockeyeSalmon` classes that were called for in Chapter 4.
Listing A-10. The Animal Class Abstracting Over Birds and Fish (and Other Organisms)

public abstract class Animal
{
    private String kind;
    private String appearance;

    public Animal(String kind, String appearance)
    {
        this.kind = kind;
        this.appearance = appearance;
    }

    public abstract void eat();
    public abstract void move();

    @Override
    public final String toString()
    {
        return kind + " -- " + appearance;
    }
}

Listing A-11. The Bird Class Abstracting Over American Robins, Domestic Canaries, and Other Kinds of Birds

public abstract class Bird extends Animal
{
    public Bird(String kind, String appearance)
    {
        super(kind, appearance);
    }

    @Override
    public final void eat()
    {
        System.out.println("eats seeds and insects");
    }

    @Override
    public final void move()
    {
        System.out.println("flies through the air");
    }
}
Listing A-12. The Fish Class Abstracting Over Rainbow Trout, Sockeye Salmon, and Other Kinds of Fish

```java
public abstract class Fish extends Animal {
    public Fish(String kind, String appearance) {
        super(kind, appearance);
    }

    @Override
    public final void eat() {
        System.out.println("eats krill, algae, and insects");
    }

    @Override
    public final void move() {
        System.out.println("swims through the water");
    }
}
```

Listing A-13. The AmericanRobin Class Denoting a Bird with a Red Breast

```java
public final class AmericanRobin extends Bird {
    public AmericanRobin() {
        super("americanrobin", "red breast");
    }
}
```

Listing A-14. The DomesticCanary Class Denoting a Bird of Various Colors

```java
public final class DomesticCanary extends Bird {
    public DomesticCanary() {
        super("domestic canary", "yellow, orange, black, brown, white, red");
    }
}
```

Listing A-15. The RainbowTrout Class Denoting a Rainbow-Colored Fish

```java
public final class RainbowTrout extends Fish {
    public RainbowTrout() {
        super("rainbowtrout", "bands of brilliant speckled multicolored " +
               "stripes running nearly the whole length of its body");
    }
}
Listing A-16. The SockeyeSalmon Class Denoting a Red-and-Green Fish

```java
public final class SockeyeSalmon extends Fish {
    public SockeyeSalmon() {
        super("sockeyesalmon", "bright red with a green head");
    }
}
```

Animal’s toString() method is declared final because it doesn’t make sense to override this method, which is complete in this example. Also, each of Bird’s and Fish’s overriding eat() and move() methods is declared final because it doesn’t make sense to override these methods in this example, which assumes that all birds eat seeds and insects; all fish eat krill, algae, and insects; all birds fly through the air; and all fish swim through the water.

The AmericanRobin, DomesticCanary, RainbowTrout, and SockeyeSalmon classes are declared final because they represent the bottom of the Bird and Fish class hierarchies, and it doesn’t make sense to subclass them.

50. Listing A-17 declares the Animals class that was called for in Chapter 4.

Listing A-17. The Animals Class Letting Animals Eat and Move

```java
public class Animals {
    public static void main(String[] args) {
        Animal[] animals = { new AmericanRobin(), new RainbowTrout(),
                            new DomesticCanary(), new SockeyeSalmon() };
        for (int i = 0; i < animals.length; i++) {
            System.out.println(animals[i]);
            animals[i].eat();
            animals[i].move();
            System.out.println();
        }
    }
}
```

51. Listings A–18 through A–20 declare the Countable interface, the modified Animal class, and the modified Animals class that were called for in Chapter 4.

Listing A-18. The Countable Interface for Use in Taking a Census of Animals

```java
public interface Countable {
    String getID();
}
```
Listing A-19. The Refactored Animal Class for Help in Census Taking

```java
public abstract class Animal implements Countable {
    private String kind;
    private String appearance;

    public Animal(String kind, String appearance) {
        this.kind = kind;
        this.appearance = appearance;
    }

    public abstract void eat();
    public abstract void move();

    @Override
    public final String toString() {
        return kind + " -- " + appearance;
    }

    @Override
    public final String getID() {
        return kind;
    }
}
```

Listing A-20. The Modified Animals Class for Carrying Out the Census

```java
public class Animals {
    public static void main(String[] args) {
        Animal[] animals = { new AmericanRobin(), new RainbowTrout(),
                             new DomesticCanary(), new SockeyeSalmon(),
                             new RainbowTrout(), new AmericanRobin() };
        for (int i = 0; i < animals.length; i++) {
            System.out.println(animals[i]);
            animals[i].eat();
            animals[i].move();
            System.out.println();
        }

        Census census = new Census();
        Countable[] countables = (Countable[]) animals;
        for (int i = 0; i < countables.length; i++) {
            census.update(countables[i].getID());
        }
    }
}
```
Chapter 5: Mastering Advanced Language Features, Part 1

1. A nested class is a class that’s declared as a member of another class or scope.

2. The four kinds of nested classes are static member classes, nonstatic member classes, anonymous classes, and local classes.

3. Nonstatic member classes, anonymous classes, and local classes are also known as inner classes.

4. The answer is false: a static member class doesn’t have an enclosing instance.

5. You instantiate a nonstatic member class from beyond its enclosing class by first instantiating the enclosing class and then prefixing the new operator with the enclosing class instance as you instantiate the enclosed class. Example: `new EnclosingClass().new EnclosedClass()`.

6. It’s necessary to declare local variables and parameters final when they are being accessed by an instance of an anonymous class or a local class.

7. The answer is true: an interface can be declared within a class or within another interface.

8. A package is a unique namespace that can contain a combination of top-level classes, other top-level types, and subpackages.

9. You ensure that package names are unique by specifying your reversed Internet domain name as the top-level package name.

10. A package statement is a statement that identifies the package in which a source file’s types are located.

11. The answer is false: you cannot specify multiple package statements in a source file.

12. An import statement is a statement that imports types from a package by telling the compiler where to look for unqualified type names during compilation.

13. You indicate that you want to import multiple types via a single import statement by specifying the wildcard character (*)

14. During a runtime search, the virtual machine reports a “no class definition found” error when it cannot find a classfile.

```java
for (int i = 0; i < Census.SIZE; i++)
    System.out.println(census.get(i));
```

15. You specify the user classpath to the virtual machine via the \texttt{-classpath} (or \texttt{-cp}) option used to start the virtual machine or, when not present, the \texttt{CLASSPATH} environment variable.

16. A constant interface is an interface that only exports constants.

17. Constant interfaces are used to avoid having to qualify their names with their classes.

18. Constant interfaces are bad because their constants are nothing more than an implementation detail that shouldn’t be allowed to leak into the class's exported interface because they might confuse the class's users (what’s the purpose of these constants?). Also, they represent a future commitment: even when the class no longer uses these constants, the interface must remain to ensure binary compatibility.

19. A static import statement is a version of the import statement that lets you import a class’s static members so that you don’t have to qualify them with their class names.

20. You specify a static import statement as \texttt{import}, followed by \texttt{static}, followed by a member access operator–separated list of package and subpackage names, followed by the member access operator, followed by a class's name, followed by the member access operator, followed by a single static member name or the asterisk wildcard, for example, \texttt{import static java.lang.Math.cos}; (import the \texttt{cos()} static method from the Math class).

21. An exception is a divergence from an application’s normal behavior.

22. Objects are superior to error codes for representing exceptions because error code Boolean or integer values are less meaningful than object names and because objects can contain information about what led to the exception. These details can be helpful to a suitable workaround. Furthermore, error codes are easy to ignore.

23. A throwable is an instance of \texttt{Throwable} or one of its subclasses.

24. The \texttt{getCause()} method returns an exception that’s wrapped inside another exception.

25. \texttt{Exception} describes exceptions that result from external factors (such as not being able to open a file) and from flawed code (such as passing an illegal argument to a method). \texttt{Error} describes virtual machine-oriented exceptions such as running out of memory or being unable to load a classfile.

26. A checked exception is an exception that represents a problem with the possibility of recovery and for which the developer must provide a workaround.
27. A runtime exception is an exception that represents a coding mistake.

28. You would introduce your own exception class when no existing exception class in the standard class library meets your needs.

29. The answer is false: you use a throws clause to identify exceptions that are thrown from a method by appending this clause to a method’s header.

30. The purpose of a try statement is to provide a scope (via its brace-delimited body) in which to present code that can throw exceptions. The purpose of a catch block is to receive a thrown exception and provide code (via its brace-delimited body) that handles that exception by providing a workaround.

31. The purpose of a finally block is to provide cleanup code that’s executed whether an exception is thrown or not.

32. Listing A-21 presents the G2D class that was called for in Chapter 5.

Listing A-21. The G2D Class with Its Matrix Nonstatic Member Class

```java
public class G2D
{
    private Matrix xform;

    public G2D()
    {
        xform = new Matrix();
        xform.a = 1.0;
        xform.e = 1.0;
        xform.i = 1.0;
    }

    private class Matrix
    {
        double a, b, c;
        double d, e, f;
        double g, h, i;
    }
}
```

33. To extend the logging package (presented in Chapter 5’s discussion of packages) to support a null device in which messages are thrown away, first introduce Listing A-22's NullDevice package-private class.

Listing A-22. Implementing the Proverbial “Bit Bucket” Class

```java
package logging;

class NullDevice implements Logger
{
    private String dstName;
}
Continue by introducing, into the LoggerFactory class, a NULLDEVICE constant and code that instantiates NullDevice with a null argument—a destination name isn’t required—when newLogger()’s dstType parameter contains this constant’s value. Check out Listing A-23.

Listing A-23. A Refactored LoggerFactory Class

```java
package logging;

public abstract class LoggerFactory {
    public final static int CONSOLE = 0;
    public final static int FILE = 1;
    public final static int NULLDEVICE = 2;

    public static Logger newLogger(int dstType, String... dstName) {
        switch (dstType) {
            case CONSOLE: return new Console(dstName.length == 0 ? null : dstName[0]);
            case FILE: return new File(dstName.length == 0 ? null : dstName[0]);
            case NULLDEVICE: return new NullDevice(null);
            default: return null;
        }
    }
}
```
34. Modifying the logging package (presented in Chapter 5’s discussion of packages) so that Logger’s `connect()` method throws a `CannotConnectException` instance when it cannot connect to its logging destination, and the other two methods each throw a `NotConnectedException` instance when `connect()` was not called or when it threw a `CannotConnectException` instance, results in Listing A-24’s Logger interface.


```java
package logging;

public interface Logger
{
    void connect() throws CannotConnectException;
    void disconnect() throws NotConnectedException;
    void log(String msg) throws NotConnectedException;
}
```

Listing A-25 presents the `CannotConnectException` class.

Listing A-25. An Uncomplicated `CannotConnectException` Class

```java
package logging;

public class CannotConnectException extends Exception
{
}
```

The `NotConnectedException` class has the same structure but with a different name. Listing A-26 presents the `Console` class.

Listing A-26. The `Console` Class Satisfying Logger’s Contract Without Throwing Exceptions

```java
package logging;

class Console implements Logger
{
    private String dstName;

    Console(String dstName)
    {
        this.dstName = dstName;
    }

    public void connect() throws CannotConnectException
    {
    }

    public void disconnect() throws NotConnectedException
    {
    }
```
public void log(String msg) throws NotConnectedException
{
    System.out.println(msg);
}

Listing A-27 presents the File class.

Listing A-27. The File Class Satisfying Logger’s Contract by Throwing Exceptions As Necessary

```java
package logging;

class File implements Logger
{
    private String dstName;

    File(String dstName)
    {
        this.dstName = dstName;
    }

    public void connect() throws CannotConnectException
    {
        if (dstName == null)
            throw new CannotConnectException();
    }

    public void disconnect() throws NotConnectedException
    {
        if (dstName == null)
            throw new NotConnectedException();
    }

    public void log(String msg) throws NotConnectedException
    {
        if (dstName == null)
            throw new NotConnectedException();
        System.out.println("writing \" + msg + \" to file \" + dstName);
    }
}
```

35. When you modify TestLogger to respond appropriately to thrown CannotConnectException and NotConnectedException objects, you end up with something similar to Listing A-28.

Listing A-28. A TestLogger Class That Handles Thrown Exceptions

```java
import logging.*;

public class TestLogger
{
    public static void main(String[] args)
    {
```

www.it-ebooks.info
```java
{ try {
    Logger logger = LoggerFactory.newLogger(LoggerFactory.CONSOLE);
    logger.connect();
    logger.log("test message #1");
    logger.disconnect();
} catch (CannotConnectException cce) {
    System.err.println("cannot connect to console-based logger");
} catch (NotConnectedException nce) {
    System.err.println("not connected to console-based logger");
}

try {
    Logger logger = LoggerFactory.newLogger(LoggerFactory.FILE, "x.txt");
    logger.connect();
    logger.log("test message #2");
    logger.disconnect();
} catch (CannotConnectException cce) {
    System.err.println("cannot connect to file-based logger");
} catch (NotConnectedException nce) {
    System.err.println("not connected to file-based logger");
}

try {
    Logger logger = LoggerFactory.newLogger(LoggerFactory.FILE);
    logger.connect();
    logger.log("test message #3");
    logger.disconnect();
} catch (CannotConnectException cce) {
    System.err.println("cannot connect to file-based logger");
} catch (NotConnectedException nce) {
    System.err.println("not connected to file-based logger");
}
}
```
Chapter 6: Mastering Advanced Language Features, Part 2

1. An assertion is a statement that lets you express an assumption of program correctness via a Boolean expression.

2. You would use assertions to validate internal invariants, control-flow invariants, preconditions, postconditions, and class invariants.

3. The answer is false: specifying the -ea command-line option with no argument enables all assertions except for system assertions.

4. An annotation is an instance of an annotation type and associates metadata with an application element. It's expressed in source code by prefixing the type name with the @ symbol.

5. Constructors, fields, local variables, methods, packages, parameters, and types (annotation, class, enum, and interface) can be annotated.

6. The three compiler-supported annotation types are Override, Deprecated, and SuppressWarnings.

7. You declare an annotation type by specifying the @ symbol, immediately followed by reserved word interface, followed by the type’s name, followed by a body.

8. A marker annotation is an instance of an annotation type that supplies no data apart from its name; the type's body is empty.

9. An element is a method header that appears in the annotation type’s body. It cannot have parameters or a throws clause. Its return type must be primitive (such as int), String, Class, an enum type, an annotation type, or an array of the preceding types. It can have a default value.

10. You assign a default value to an element by specifying default followed by the value, whose type must match the element’s return type. For example, String developer() default "unassigned";

11. A meta-annotation is an annotation that annotates an annotation type.

12. Java’s four meta-annotation types are Target, Retention, Documented, and Inherited.

13. Generics can be defined as a suite of language features for declaring and using type-agnostic classes and interfaces.

14. You would use generics to ensure that your code is typesafe by avoiding thrown ClassCastExceptions.

15. The difference between a generic type and a parameterized type is that a generic type is a class or interface that introduces a family of parameterized types by declaring a formal type parameter list, and a parameterized type is an instance of a generic type.
16. Anonymous classes cannot be generic because they have no names.

17. The five kinds of actual type arguments are concrete types, concrete parameterized types, array types, type parameters, and wildcards.

18. The answer is true: you cannot specify a primitive-type name (such as `double` or `int`) as an actual type argument.

19. A raw type is a generic type without its type parameters.

20. The compiler reports an unchecked warning message when it detects an explicit cast that involves a type parameter. The compiler is concerned that downcasting to whatever type is passed to the type parameter might result in a violation of type safety.

21. You suppress an unchecked warning message by prefixing the constructor or method that contains the unchecked code with the `@SuppressWarnings("unchecked")` annotation.

22. The answer is true: `List<E>`'s `E` type parameter is unbounded.

23. You specify a single upper bound via reserved word `extends` followed by a type name.

24. A recursive type bound is a type parameter bound that includes the type parameter.

25. Wildcard type arguments are necessary because by accepting any actual type argument they provide a typesafe workaround to the problem of polymorphic behavior not applying to multiple parameterized types that differ only in regard to one type parameter being a subtype of another type parameter. For example, because `List<String>` isn’t a kind of `List<Object>`, you cannot pass an object whose type is `List<String>` to a method parameter whose type is `List<Object>`. However, you can pass a `List<String>` object to `List<?>`, provided that you’re not going to add the `List<String>` object to the `List<?>`.

26. A generic method is a class or instance method with a type-generalized implementation.

27. Although you might think otherwise, Listing 6–36’s `methodCaller()` generic method calls `someOverloadedMethod(Object o)`. This method, instead of `someOverloadedMethod(Date d)`, is called because overload resolution happens at compile time, when the generic method is translated to its unique bytecode representation, and erasure (which takes care of that mapping) causes type parameters to be replaced by their leftmost bound or `Object` (when there is no bound). After erasure, you are left with Listing A-29’s nongeneric `methodCaller()` method.
Listing A-29. The Nongeneric methodCaller() Method That Results from Erasure

```java
public static void methodCaller(Object t) {
    someOverloadedMethod(t);
}
```

28. Reification is representing the abstract as if it was concrete.

29. The answer is false: type parameters are not reified.

30. Erasure is the throwing away of type parameters following compilation so that they are not available at runtime. Erasure also involves replacing uses of other type variables by the upper bound of the type variable (such as Object) and inserting casts to the appropriate type when the resulting code isn’t type correct.

31. An enumerated type is a type that specifies a named sequence of related constants as its legal values.

32. Three problems that can arise when you use enumerated types whose constants are int-based are lack of compile-time type safety, brittle applications, and the inability to translate int constants into meaningful string-based descriptions.

33. An enum is an enumerated type that’s expressed via reserved word enum.

34. You use a switch statement with an enum by specifying an enum constant as the statement’s selector expression and constant names as case values.

35. You can enhance an enum by adding fields, constructors, and methods; you can even have the enum implement interfaces. Also, you can override toString() to provide a more useful description of a constant’s value and subclass constants to assign different behaviors.

36. The purpose of the abstract Enum class is to serve as the common base class of all Java language-based enumeration types.

37. The difference between Enum’s name() and toString() methods is that name() always returns a constant’s name, but toString() can be overridden to return a more meaningful description instead of the constant’s name.

38. The answer is true: Enum’s generic type is Enum<E extends Enum<E>>.

39. Listing A-30 presents a ToDo marker annotation type that annotates only type elements and that also uses the default retention policy.
Listing A-30. The ToDo Annotation Type for Marking Types That Need to Be Completed

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
public @interface ToDo
{
}

40. Listing A-31 presents a rewritten StubFinder application that works with Listing 6–13’s Stub annotation type (with appropriate @Target and @Retention annotations) and Listing 6–14’s Deck class.

Listing A-31. Reporting a Stub’s ID, Due Date, and Developer via a New Version of StubFinder

import java.lang.reflect.Method;

public class StubFinder
{
    public static void main(String[] args) throws Exception
    {
        if (args.length != 1)
        {
            System.err.println("usage: java StubFinder classfile");
            return;
        }
        Method[] methods = Class.forName(args[0]).getMethods();
        for (int i = 0; i < methods.length; i++)
        {
            if (methods[i].isAnnotationPresent(Stub.class))
            {
                Stub stub = methods[i].getAnnotation(Stub.class);
                System.out.println("Stub ID = " + stub.id());
                System.out.println("Stub Date = " + stub.dueDate());
                System.out.println("Stub Developer = " + stub.developer());
                System.out.println();
            }
        }
    }
}

41. Listing A-32 presents the generic Stack class and the StackEmptyException and StackFullException helper classes that were called for in Chapter 6.

Listing A-32. Stack and Its StackEmptyException and StackFullException Helper Classes Proving That Not All Helper Classes Need to Be Nested

public class Stack<E>
{
    private E[] elements;
    private int top;
}
@SuppressWarnings("unchecked")
Stack(int size)
{
    if (size < 2)
        throw new IllegalArgumentException("" + size);
    elements = (E[]) new Object[size];
    top = -1;
}

void push(E element) throws StackFullException
{
    if (top == elements.length - 1)
        throw new StackFullException();
    elements[++top] = element;
}

E pop() throws StackEmptyException
{
    if (isEmpty())
        throw new StackEmptyException();
    return elements[top--];
}

boolean isEmpty()
{
    return top == -1;
}

public static void main(String[] args)
    throws StackFullException, StackEmptyException
{
    Stack<String> stack = new Stack<String>(5);
    assert stack.isEmpty();
    stack.push("A");
    stack.push("B");
    stack.push("C");
    stack.push("D");
    stack.push("E");
    // Uncomment the following line to generate a StackFullException.
    // stack.push("F");
    while (!stack.isEmpty())
        System.out.println(stack.pop());
    // Uncomment the following line to generate a StackEmptyException.
    // stack.pop();
    assert stack.isEmpty();
}
class StackEmptyException extends Exception
{
}

class StackFullException extends Exception
{
}

42. Listing A-33 presents the Compass enum that was called for in Chapter 6.

Listing A-33. A Compass Enum with Four Direction Constants

class Compass
{
    NORTH, SOUTH, EAST, WEST
}

Listing A-34 presents the UseCompass class that was called for in Chapter 6.

Listing A-34. Using the Compass Enum to Keep from Getting Lost

class UseCompass
{
    public static void main(String[] args)
    {
        int i = (int) (Math.random() * 4);
        Compass[] dir = { Compass.NORTH, Compass.EAST, Compass.SOUTH, Compass.WEST };
        switch(dir[i])
        {
            case NORTH: System.out.println("heading north"); break;
            case EAST : System.out.println("heading east"); break;
            case SOUTH: System.out.println("heading south"); break;
            case WEST : System.out.println("heading west"); break;
            default   : assert false; // Should never be reached.
        }
    }
}

Chapter 7: Exploring the Basic APIs, Part 1

1. Math declares double constants $E$ and $\pi$ that represent, respectively, the natural logarithm base value ($2.71828.\ldots$) and the ratio of a circle’s circumference to its diameter ($3.14159.\ldots$). $E$ is initialized to 2.718281828459045 and $\pi$ is initialized to 3.141592653589793.

2. Math.abs(Integer.MIN_VALUE) equals Integer.MIN_VALUE because there doesn’t exist a positive 32-bit integer equivalent of MIN_VALUE. (Integer.MIN_VALUE equals -2147483648 and Integer.MAX_VALUE equals 2147483647.)
3. Math's random() method returns a pseudorandom number between 0.0 (inclusive) and 1.0 (exclusive). The expression (int) Math.random() * limit is incorrect because this expression always returns 0. The (int) cast operator has higher precedence than *, which means that the cast is performed before multiplication. random() returns a fractional value and the cast converts this value to 0, which is then multiplied by limit's value, resulting in an overall value of 0.

4. The five special values that can arise during floating-point calculations are +infinity, -infinity, NaN, +0.0, and -0.0.

5. Math and StrictMath differ in the following ways:
   a. StrictMath's methods return exactly the same results on all platforms. In contrast, some of Math's methods might return values that vary ever so slightly from platform to platform.
   b. Because StrictMath cannot utilize platform-specific features such as an extended-precision math coprocessor, an implementation of StrictMath might be less efficient than an implementation of Math.

6. The purpose of strictfp is to restrict floating-point calculations to ensure portability. This reserved word accomplishes portability in the context of intermediate floating-point representations and overflows/underflows (generating a value too large or small to fit a representation). Furthermore, it can be applied at the method level or at the class level.

7. BigDecimal is an immutable class that represents a signed decimal number (such as 23.653) of arbitrary precision with an associated scale. You might use this class to store floating-point values accurately, which represent monetary values and properly round the result of each monetary calculation.

8. The RoundingMode constant that describes the form of rounding commonly taught at school is HALF_UP.

9. BigInteger is an immutable class that represents a signed integer of arbitrary precision. It stores its value in two's complement format.

10. A primitive type wrapper class is a class whose instances wrap themselves around primitive-type values.

11. Java's primitive type wrapper classes are Boolean, Byte, Character, Double, Float, Integer, Long, and Short.

12. Java provides primitive type wrapper classes to facilitate storing primitive-type values in collections and as a convenient place to associate useful constants and class methods with the primitive-types.

13. The answer is false: Boolean is the smallest of the primitive type wrapper classes.
14. You should use Character class methods instead of expressions such as 
   \( ch \geq '0' \&\& ch \leq '9' \) to determine whether or not a character is a digit, 
a letter, and so on because it’s too easy to introduce a bug into the 
expression, expressions are not very descriptive of what they’re testing, and 
the expressions are biased toward Latin digits (0–9) and letters (A–Z and a–z).

15. You determine whether or not double variable \( d \) contains +infinity or -infinity 
   by passing this variable as an argument to Double’s boolean 
isInfinite(double \( d \)) class method, which returns true when this argument 
is +infinity or -infinity.

16. Number is the superclass of Byte, Character, and the other primitive type 
    wrapper classes.

17. The answer is true: a string literal is a String object.

18. The purpose of String’s intern() method is to store a unique copy of a 
    String object in an internal table of String objects. intern() makes it 
    possible to compare strings via their references and == or !=. These 
    operators are the fastest way to compare strings, which is especially valuable 
    when sorting a huge number of strings.

19. String and StringBuffer differ in that String objects contain immutable 
    sequences of characters, whereas StringBuffer objects contain mutable 
    sequences of characters.

20. StringBuffer and StringBuilder differ in that StringBuffer methods are 
    synchronized, whereas StringBuilder’s equivalent methods are not 
    synchronized. As a result, you would use the thread-safe but slower 
    StringBuffer class in multithreaded situations and the nonthread-safe but 
    faster StringBuilder class in single-threaded situations.

21. You invoke the System.arraycopy() method to copy an array to another array.

22. You invoke the System.currentTimeMillis() method to obtain the current 
    time in milliseconds.

23. A thread is an independent path of execution through an application’s code.

24. The purpose of the Runnable interface is to identify those objects that supply 
    code for threads to execute via this interface’s solitary void run() method.

25. The purpose of the Thread class is to provide a consistent interface to the 
    underlying operating system’s threading architecture. It provides methods 
    that make it possible to associate code with threads as well as to start and 
    manage those threads.

26. The answer is false: a Thread object associates with a single thread.
27. A race condition is a scenario in which multiple threads are accessing shared data, and the final result of these accesses is dependent on the timing of how the threads are scheduled. Race conditions can lead to bugs that are hard to find and results that are unpredictable.

28. Synchronization is the act of allowing only one thread at a time to execute code within a method or a block.

29. Synchronization is implemented in terms of monitors and locks.

30. Synchronization works by requiring that a thread that wants to enter a monitor-controlled critical section first acquire a lock. The lock is released automatically when the thread exits the critical section.

31. The answer is true: variables of type long or double are not atomic on 32-bit virtual machines.

32. The purpose of reserved word volatile is to let threads running on multiprocessor or multicore machines access the main memory copy of an instance field or class field. Without volatile, each thread might access its cached copy of the field and won't see modifications made by other threads to their copies.

33. The answer is false: Object's wait() methods cannot be called from outside of a synchronized method or block.

34. Deadlock is a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section.

35. The purpose of the ThreadLocal class is to associate per-thread data (such as a user ID) with a thread.

36. InheritableThreadLocal differs from ThreadLocal in that the former class lets a child thread inherit a thread-local value from its parent thread.

37. Listing A-35 presents the PrimeNumberTest application that was called for in Chapter 7.

Listing A-35. Checking a Positive Integer Argument to Discover If It's Prime

```java
public class PrimeNumberTest
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java PrimeNumberTest integer");
            System.err.println("integer must be 2 or higher");
            return;
        }
```
try {
    int n = Integer.parseInt(args[0]);
    if (n < 2) {
        System.err.println(n + " is invalid because it is less than 2");
        return;
    }
    for (int i = 2; i <= Math.sqrt(n); i++)
        if (n % i == 0) {
            System.out.println(n + " is not prime");
            return;
        }
    System.out.println(n + " is prime");
} catch (NumberFormatException nfe) {
    System.err.println("unable to parse " + args[0] + " into an int");
}
}

38. Listing A-36 presents the MultiPrint application that was called for in Chapter 7.

Listing A-36. Printing a Line of Text Multiple Times

public class MultiPrint {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java MultiPrint text count");
            return;
        }
        String text = args[0];
        int count = Integer.parseInt(args[1]);
        for (int i = 0; i < count; i++)
            System.out.println(text);
    }
}

39. The following loop uses StringBuffer to minimize object creation:

String[] imageNames = new String[NUM IMAGES];
StringBuffer sb = new StringBuffer();
for (int i = 0; i < imageNames.length; i++) {
    sb.append("image");
    sb.append(i);
sb.append(".png");
imageNames[i] = sb.toString();
sb.setLength(0); // Erase previous StringBuffer contents.
}

40. Listing A-37 presents the DigitsToWords application that was called for in Chapter 7.

Listing A-37. Converting an Integer Value to Its Textual Representation

public class DigitsToWords
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java DigitsToWords integer");
            return;
        }
        System.out.println(convertDigitsToWords(Integer.parseInt(args[0])));
    }

    static String convertDigitsToWords(int integer)
    {
        if (integer < 0 || integer > 9999)
            throw new IllegalArgumentException("Out of range: " + integer);
        if (integer == 0)
            return "zero";
        String[] group1 =
        {
            "one",
            "two",
            "three",
            "four",
            "five",
            "six",
            "seven",
            "eight",
            "nine"
        };
        String[] group2 =
        {
            "ten",
            "eleven",
            "twelve",
            "thirteen",
            "fourteen",
            "fifteen",
            "sixteen",
            "seventeen",
            "eighteen",
            "nineteen",
            "twenty",
            "thirty",
            "forty",
            "fifty",
            "sixty",
            "seventy",
            "eighty",
            "ninety"
        };
        return group1[integer / 100] + " " + group2[integer % 100 - 1] + (integer == 0 ? "" : "-");
    }
}
"eighteen",
"nineteen"
};
String[] group3 =
{
"twenty",
"thirty",
"fourty",
"fifty",
"sixty",
"seventy",
"eighty",
"ninety"
};
StringBuffer result = new StringBuffer();
if (integer >= 1000)
{
    int tmp = integer / 1000;
    result.append(group1[tmp - 1] + " thousand");
    integer -= tmp * 1000;
    if (integer == 0)
        return result.toString();
    result.append(" ");
}
if (integer >= 100)
{
    int tmp = integer / 100;
    result.append(group1[tmp - 1] + " hundred");
    integer -= tmp * 100;
    if (integer == 0)
        return result.toString();
    result.append(" and ");
}
if (integer >= 10 && integer <= 19)
{
    result.append(group2[integer - 10]);
    return result.toString();
}
if (integer >= 20)
{
    int tmp = integer / 10;
    result.append(group3[tmp - 2]);
    integer -= tmp * 10;
    if (integer == 0)
        return result.toString();
    result.append("-");
}    result.append(group1[integer - 1]);
    return result.toString();
}
41. Listing A-38 presents the EVDump application that was called for in Chapter 7.

Listing A-38. Dumping All Environment Variables to Standard Output

```java
public class EVDump {
    public static void main(String[] args) {
        System.out.println(System.getenv()); // System.out.println() calls toString()
        // on its object argument and outputs this
        // string
    }
}
```

42. Listing A-39 presents the revised CountingThreads application that was called for in Chapter 7.

Listing A-39. Counting via Daemon Threads

```java
public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true) {
                    System.out.println(name + " : " + count++);
                }
            }
        };
        Thread thdA = new Thread(r);
        thdA.setDaemon(true);
        Thread thdB = new Thread(r);
        thdB.setDaemon(true);
        thdA.start();
        thdB.start();
    }
}
```

When you run this application, the two daemon threads start executing, and you will probably see some output. However, the application will end as soon as the default main thread leaves the `main()` method and dies.

43. Listing A-40 presents the StopCountingThreads application that was called for in Chapter 7.
Listing A-40. Stopping the Counting Threads When Return/Enter is Pressed

```java
import java.io.IOException;

public class StopCountingThreads {
    private static volatile boolean stopped = false;

    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (!stopped)
                    System.out.println(name + ":: " + count++);
            }
        }
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
        try { System.in.read(); } catch (IOException ioe) {} 
        stopped = true;
    }
}
```

Chapter 8: Exploring the Basic APIs, Part 2

1. Instances of the `Random` class generate sequences of random numbers by starting with a special 48-bit value that’s known as a seed. This value is subsequently modified by a mathematical algorithm, which is known as a linear congruential generator.

2. The root set of references is a collection of local variables, parameters, class fields, and instance fields that currently exist and that contain (possibly null) references to objects.

3. The answer is false: the garbage collector can collect unreachable objects only.

4. The different levels of reachability are strongly reachable, softly reachable, weakly reachable, and phantom reachable.

5. The difference between a soft reference and a weak reference is that the garbage collector is more eager to collect an object that is reachable only via a weak reference.
6. The classes that comprise the References API are Reference, ReferenceQueue, SoftReference, WeakReference, and PhantomReference. All of these classes are located in the java.lang.ref package.

7. You would use the SoftReference class to implement caches of objects that are expensive timewise to create.

8. You would use the PhantomReference class as a replacement for the finalize() method.

9. Some of the capabilities offered by the Reflection API are letting applications dynamically load and learn about loaded classes and other reference types; and letting applications instantiate classes, call methods, access fields, and perform other tasks reflectively.

10. Reflection shouldn’t be used indiscriminately for several reasons. Application performance suffers because it takes longer to perform operations with reflection than without reflection. Also, reflection-oriented code can be harder to read, and the absence of compile-time type checking can result in runtime failures.

11. The class that’s the entry point into the Reflection API is java.lang.Class.

12. The answer is false: not all of the Reflection API is contained in the java.lang.reflect package. For example, Class is located in the java.lang package.

13. The three ways to obtain a Class object are to invoke Class’s forName() method, to invoke Object’s getClass() method, and to use a class literal.

14. The answer is true: you can use class literals with primitive-types.

15. You instantiate a dynamically loaded class by invoking Class’s newInstance() method.

16. You invoke Constructor’s Class[]<?> getParameterTypes() method to obtain a constructor’s parameter types.

17. Class’s Field getField(String name) method throws NoSuchFieldException when it cannot locate the named field.

18. You determine if a method is declared to receive a variable number of arguments by invoking Method’s isVarArgs() method on the Method object that represents the method.

19. The answer is true: you can reflectively make a private method accessible. You do this by invoking the setAccessible() method that each of Constructor, Field, and Method inherits from its AccessibleObject superclass.
20. The purpose of Package's `isSealed()` method is to indicate whether or not a package is sealed (all classes that are part of the package are archived in the same JAR file). This method returns true when the package is sealed.

21. The answer is true: `getPackage()` requires at least one classfile to be loaded from the package before it returns a Package object describing that package.

22. You reflectively create and access a Java array by invoking the class methods declared in the `java.lang.reflect.Array` class.

23. The purpose of the `StringTokenizer` class is to provide access to a string's individual components.

24. The `java.util` package's `Timer` and `TimerTask` classes are the standard class library's convenient and simpler alternative to the Threads API for scheduling task execution.

25. The answer is false: `Timer()` creates a new timer whose task-execution thread doesn’t run as a daemon thread.

26. In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. When an execution is delayed for any reason (such as garbage collection), subsequent executions are also delayed.

27. You call the `schedule()` methods to schedule a task for fixed-delay execution.

28. In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. When an execution is delayed for any reason (such as garbage collection), two or more executions will occur in rapid succession to “catch up.”

29. The difference between `Timer`'s `cancel()` method and `TimerTask`'s `cancel()` method is as follows: `Timer`'s `cancel()` method terminates the timer, discarding any currently scheduled timer tasks. In contrast, `TimerTask`'s `cancel()` method cancels the invoking timer task only.

30. Listing A-41 presents the `Guess` application that was called for in Chapter 8.

Listing A-41. Guessing Game

```java
import java.util.Random;

public class Guess
{
    public static void main(String[] args) throws java.io.IOException
    {
        Random r = new Random();
        int hiddenValue = 'a' + r.nextInt(26);

        while (true)
```

www.it-ebooks.info
int guess = 0;
while (guess < 'a' || guess > 'z')
{
    System.out.print("Guess between a and z inclusive: ");
    guess = System.in.read();
    System.out.println();

    // Flush carriage return or carriage return/newline combination
    // so that each character isn't automatically read during the
    // next System.in.read() method call.

    int x = 0;
    while (x != '\n')
    {
        x = System.in.read();
    }
}
if (guess < hiddenValue)
    System.out.println("too low");
else
    if (guess > hiddenValue)
        System.out.println("too high");
    else
    {
        System.out.println("you got it");
        break;
    }
}

31. Listing A-42 presents the Classify application that was called for in Chapter 8.

Listing A-42. Classifying a Command-Line Argument as an Annotation Type, Enum, Interface, or Class

public class Classify
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java Classify pkgAndTypeName");
            return;
        }

        try
        {
            Class<?> clazz = Class.forName(args[0]);
            if (clazz.isAnnotation())
                System.out.println("Annotation");
            else
                if (clazz.isEnum())
                    System.out.println("Enum");
        }
    }
}
else
    if (clazz.isInterface())
        System.out.println("Interface");
    else
        System.out.println("Class");
}
catch (ClassNotFoundException cnfe)
{
    System.err.println("could not locate " + args[0]);
}

Specify java Classify java.lang.Override, and you'll see Annotation as the output. Also, java.Classify java.math.RoundingMode outputs Enum, java Classify java.lang. Runnable outputs Interface, and java Classify java.lang.Class outputs Class.

32. Listing A-43 presents the Tokenize application that was called for in Chapter 8.

Listing A-43. Extracting the Month, Day, Year, Hour, Minute, and Second Tokens from a Date String

import java.util.StringTokenizer;

public class Tokenize
{
    public static void main(String[] args)
    {
        String date = "03-12-2014 03:05:20";
        StringTokenizer st = new StringTokenizer(date, "- :");
        System.out.println("Month = " + st.nextToken());
        System.out.println("Day = " + st.nextToken());
        System.out.println("Year = " + st.nextToken());
        System.out.println("Hour = " + st.nextToken());
        System.out.println("Minute = " + st.nextToken());
        System.out.println("Second = " + st.nextToken());
    }
}

33. Listing A-44 presents the BackAndForth application that was called for in Chapter 8.

Listing A-44. Repeatedly Moving an Asterisk Back and Forth via a Timer

import java.util.Timer;
import java.util.TimerTask;

public class BackAndForth
{
    static enum Direction { FORWARDS, BACKWARDS }

    public static void main(String[] args)
    {

TimerTask task = new TimerTask()
{
    final static int MAXSTEPS = 20;

    Direction direction = Direction.FORWARDS;

    int steps = 0;

    @Override
    public void run()
    {
        switch (direction)
        {
            case FORWARDS :
                System.out.print("\b ");
                System.out.print("*");
                break;

            case BACKWARDS:
                System.out.print("\b ");
                System.out.print("\b\b*");
        }

        if (++steps == MAXSTEPS)
        {
            direction = (direction == Direction.FORWARDS)
                ? Direction.BACKWARDS
                : Direction.FORWARDS;
            steps = 0;
        }
    }

    Timer timer = new Timer();
    timer.schedule(task, 0, 100);
}

Chapter 9: Exploring the Collections Framework

1. A collection is a group of objects that are stored in an instance of a class designed for this purpose.

2. The Collections Framework is a group of types that offers a standard architecture for representing and manipulating collections.

3. The Collections Framework largely consists of core interfaces, implementation classes, and utility classes.

4. A comparable is an object whose class implements the Comparable interface.

5. You would have a class implement the Comparable interface when you want objects to be compared according to their natural ordering.
6. A comparator is an object whose class implements the Comparator interface. Its purpose is to allow objects to be compared according to an order that’s different from their natural ordering.

7. The answer is false: a collection uses a comparable (an object whose class implements the Comparable interface) to define the natural ordering of its elements.

8. The Iterable interface describes any object that can return its contained objects in some sequence.

9. The Collection interface represents a collection of objects that are known as elements.

10. A situation where Collection’s add() method would throw an instance of the UnsupportedOperationException class is an attempt to add an element to an unmodifiable collection.

11. Iterable’s iterator() method returns an instance of a class that implements the Iterator interface. This interface provides a hasNext() method to determine if the end of the iteration over the collection has been reached, a next() method to return a collection’s next element, and a remove() method to remove the last element returned by next() from the collection.

12. The purpose of the enhanced for loop statement is to simplify collection or array iteration.

13. The enhanced for loop statement is expressed as for (typeid: collection) or for (typeid: array) and reads “for each type object in collection, assign this object to id at the start of the loop iteration” or “for each type object in array, assign this object to id at the start of the loop iteration.”

14. The answer is true: the enhanced for loop works with arrays. For example, int[] x = { 1, 2, 3 }; for (int i: x) System.out.println(i); declares array x and outputs all of its int-based elements.

15. Autoboxing is the act of wrapping a primitive-type value in an object of a primitive type wrapper class whenever a primitive-type is specified but a reference is required. This feature saves the developer from having to instantiate a wrapper class explicitly when storing the primitive-type value in a collection.

16. Unboxing is the act of unwrapping a primitive-type value from its wrapper object whenever a reference is specified but a primitive-type is required. This feature saves the developer from having to call a method explicitly on the object (such as intValue()) to retrieve the wrapped value.

17. A list is an ordered collection, which is also known as a sequence. Elements can be stored in and accessed from specific locations via integer indexes.
18. A `ListIterator` instance uses a cursor to navigate through a list.

19. A view is a list that's backed by another list. Changes that are made to the view are reflected in this backing list.

20. You would use the `subList()` method to perform range-view operations over a collection in a compact manner. For example, `list.subList(fromIndex, toIndex).clear();` removes a range of elements from `list`, where the first element is located at `fromIndex` and the last element is located at `toIndex - 1`.

21. The `ArrayList` class provides a list implementation that's based on an internal array.

22. The `LinkedList` class provides a list implementation that's based on linked nodes.

23. A node is a fixed sequence of value and link memory locations (that is, an arrangement of a specific number of values and links, such as one value location followed by one link location). From an object-oriented perspective, it's an object whose fields store values and references to other node objects. These references are also known as links.

24. The answer is false: `ArrayList` provides slower element insertions and deletions than `LinkedList`.

25. A set is a collection that contains no duplicate elements.

26. The `TreeSet` class provides a set implementation that's based on a tree data structure. As a result, elements are stored in sorted order.

27. The `HashSet` class provides a set implementation that's backed by a hashtable data structure.

28. The answer is true: to avoid duplicate elements in a `HashSet`, your own classes must correctly override `equals()` and `hashCode()`.

29. The difference between `HashSet` and `LinkedHashSet` is that `LinkedHashSet` uses a linked list to store its elements, resulting in its iterator returning elements in the order in which they were inserted.

30. The `EnumSet` class provides a `Set` implementation that's based on a bitset.

31. A sorted set is a set that maintains its elements in ascending order, sorted according to their natural ordering or according to a comparator that's supplied when the sorted set is created. Furthermore, the set's implementation class must implement the `SortedSet` interface.

32. A navigable set is a sorted set that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets.
33. The answer is false: HashSet isn’t an example of a sorted set. However, TreeSet is an example of a sorted set.

34. A sorted set’s add() method would throw ClassCastException when you attempt to add an element to the sorted set because the element’s class doesn’t implement Comparable.

35. A queue is a collection in which elements are stored and retrieved in a specific order. Most queues are categorized as “first-in, first out,” “last-in, first-out,” or priority.

36. The answer is true: Queue’s element() method throws NoSuchElementException when it’s called on an empty queue.

37. The PriorityQueue class provides an implementation of a priority queue, which is a queue that orders its elements according to their natural ordering or by a comparator provided when the queue is instantiated.

38. A map is a group of key/value pairs (also known as entries).

39. The TreeMap class provides a map implementation that’s based on a red-black tree. As a result, entries are stored in sorted order of their keys.

40. The HashMap class provides a map implementation that’s based on a hashtable data structure.

41. A hashtable uses a hash function to map keys to integer values.

42. Continuing from the previous exercise, the resulting integer values are known as hash codes. They identify hashtable array elements, which are known as buckets or slots.

43. A hashtable’s capacity refers to the number of buckets.

44. A hashtable’s load factor refers to the ratio of the number of stored entries divided by the number of buckets.

45. The difference between HashMap and LinkedHashMap is that LinkedHashMap uses a linked list to store its entries, resulting in its iterator returning entries in the order in which they were inserted.

46. The IdentityHashMap class provides a Map implementation that uses reference equality (==) instead of object equality (equals()) when comparing keys and values.

47. The EnumMap class provides a Map implementation whose keys are the members of the same enum.

48. A sorted map is a map that maintains its entries in ascending order, sorted according to the keys’ natural ordering or according to a comparator that’s supplied when the sorted map is created. Furthermore, the map’s implementation class must implement the SortedMap interface.
49. A navigable map is a sorted map that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets.

50. The answer is true: TreeMap is an example of a sorted map.

51. The purpose of the Arrays class’s static \(<T>\) List\(\langle T \rangle\) asList(T... array) method is to return a fixed-size list backed by the specified array. (Changes to the returned list “write through” to the array.)

52. The answer is false: binary search is faster than linear search.

53. You would use Collections’ static \(<T>\) Set\(\langle T \rangle\) synchronizedSet(Set\(\langle T \rangle\) s) method to return a synchronized variation of a HashSet.

54. The seven legacy collections-oriented types are Vector, Enumeration, Stack, Dictionary, Hashtable, Properties, and BitSet.

55. Listing A-45 presents the JavaQuiz application that was called for in Chapter 9.

Listing A-45. How Much Do You Know About Java? Take the Quiz and Find Out!

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class JavaQuiz
{
    private static class QuizEntry
    {
        private String question;
        private String[] choices;
        private char answer;

        QuizEntry(String question, String[] choices, char answer)
        {
            this.question = question;
            this.choices = choices;
            this.answer = answer;
        }

        String[] getChoices()
        {
            // Demonstrate returning a copy of the choices array to prevent clients
            // from directly manipulating (and possibly screwing up) the internal
            // choices array.
            String[] temp = new String[choices.length];
            System.arraycopy(choices, 0, temp, 0, choices.length);
            return temp;
        }
    }
}
```
String getQuestion()
{
    return question;
}

char getAnswer()
{
    return answer;
}
}

static QuizEntry[] quizEntries =
{
    new QuizEntry("What was Java's original name?",
        new String[] { "Oak", "Duke", "J", "None of the above" },
        'A'),
    new QuizEntry("Which of the following reserved words is also a literal?",
        new String[] { "for", "long", "true", "enum" },
        'C'),
    new QuizEntry("The conditional operator (?:) resembles which statement?",
        new String[] { "switch", "if-else", "if", "while" },
        'B')
};

public static void main(String[] args)
{
    // Populate the quiz list.
    List<QuizEntry> quiz = new ArrayList<QuizEntry>();
    for (QuizEntry entry : quizEntries)
        quiz.add(entry);
    // Perform the quiz.
    System.out.println("Java Quiz");
    System.out.println("--------
");
    Iterator<QuizEntry> iter = quiz.iterator();
    while (iter.hasNext())
    {
        QuizEntry qe = iter.next();
        System.out.println(qe.getQuestion());
        String[] choices = qe.getChoices();
        for (int i = 0; i < choices.length; i++)
            System.out.println("  " + (char) ('A' + i) + ": " + choices[i]);
        int choice = -1;
        while (choice < 'A' || choice > 'A' + choices.length)
        {
            System.out.print("Enter choice letter: ");
            try
            {
                choice = System.in.read();
                // Remove trailing characters up to and including the newline
                // to avoid having these characters automatically returned in
                // subsequent System.in.read() method calls.
                while (System.in.read() != '\n');
            }
            catch (IOException e) {}
choice = Character.toUpperCase((char) choice);
}
catch (java.io.IOException ioe)
{
}

if (choice == qe.getAnswer())
    System.out.println("You are correct!\n");
else
    System.out.println("You are not correct!\n");

56. \((int) (f ^ (f >>> 32))\) is used instead of \((int) (f ^ (f >> 32))\) in the hash code generation algorithm because >>> always shifts a 0 to the right, which doesn't affect the hash code, whereas >> shifts a 0 or a 1 to the right (whatever value is in the sign bit), which affects the hash code when a 1 is shifted.

57. Listing A-46 presents the FrequencyDemo application that was called for in Chapter 9.

Listing A-46. Reporting the Frequency of Last Command-Line Argument Occurrences in the Previous Command-Line Arguments

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
public class FrequencyDemo
{
    public static void main(String[] args)
    {
        List<String> listOfArgs = new LinkedList<String>();
        String lastArg = (args.length == 0) ? null : args[args.length - 1];
        for (int i = 0; i < args.length - 1; i++)
            listOfArgs.add(args[i]);
        System.out.println("Number of occurrences of " + lastArg + " = " +
            Collections.frequency(listOfArgs, lastArg));
    }
}

Chapter 10: Exploring the Concurrency Utilities

1. Concurrency Utilities is a framework of classes and interfaces that overcome problems with the Threads API. Specifically, low-level concurrency primitives such as synchronized and wait()/notify() are often hard to use correctly; too much reliance on the synchronized primitive can lead to performance issues, which affect an application's scalability; and higher-level constructs such as thread pools and semaphores aren't included with Java's low-level threading capabilities.
2. The packages in which Concurrency Utilities types are stored are java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks.

3. A task is an object whose class implements the Runnable interface (a runnable task) or the Callable interface (a callable task).

4. An executor is an object whose class directly or indirectly implements the Executor interface, which decouples task submission from task-execution mechanics.

5. The Executor interface focuses exclusively on Runnable, which means that there’s no convenient way for a runnable task to return a value to its caller (because Runnable’s run() method doesn’t return a value); Executor doesn’t provide a way to track the progress of executing runnable tasks, cancel an executing runnable task, or determine when the runnable task finishes execution; Executor cannot execute a collection of runnable tasks; and Executor doesn’t provide a way for an application to shut down an executor (much less to shut down an executor properly).

6. Executor’s limitations are overcome by providing the ExecutorService interface.

7. The differences existing between Runnable’s run() method and Callable’s call() method are as follows: run() cannot return a value, whereas call() can return a value; and run() cannot throw checked exceptions, whereas call() can throw checked exceptions.

8. The answer is false: you can throw checked and unchecked exceptions from Callable’s call() method but can only throw unchecked exceptions from Runnable’s run() method.

9. A future is an object whose class implements the Future interface. It represents an asynchronous computation and provides methods for canceling a task, for returning a task’s value, and for determining whether or not the task has finished.

10. The Executors class’s newFixedThreadPool() method creates a thread pool that reuses a fixed number of threads operating off of a shared unbounded queue. At most, nThreads threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue for an available thread. If any thread terminates because of a failure during execution before the executor shuts down, a new thread will take its place when needed to execute subsequent tasks. The threads in the pool will exist until the executor is explicitly shut down.

11. A synchronizer is a class that facilitates a common form of synchronization.
12. Four commonly used synchronizers are countdown latches, cyclic barriers, exchangers, and semaphores. A countdown latch lets one or more threads wait at a “gate” until another thread opens this gate, at which point these other threads can continue. A cyclic barrier lets a group of threads wait for each other to reach a common barrier point. An exchanger lets a pair of threads exchange objects at a synchronization point. A semaphore maintains a set of permits for restricting the number of threads that can access a limited resource.

13. The concurrency-oriented extensions to the Collections Framework provided by the Concurrency Utilities are ArrayBlockingQueue, BlockingDeque, BlockingQueue, ConcurrentHashMap, ConcurrencyMap, ConcurrenctNavigableMap, ConcurrentLinkedQueue, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue, and SynchronousQueue.

14. A lock is an instance of a class that implements the Lock interface, which provides more extensive locking operations than can be achieved via the synchronized reserved word. Lock also supports a wait/notification mechanism through associated Condition objects.

15. The biggest advantage that Lock objects hold over the implicit locks that are obtained when threads enter critical sections (controlled via the synchronized reserved word) is their ability to back out of an attempt to acquire a lock.

16. You obtain a Condition instance for use with a particular Lock instance by invoking Lock’s Condition newCondition() method.

17. An atomic variable is an instance of a class that encapsulates a single variable and supports lock-free, thread-safe operations on that variable, for example, AtomicInteger.

18. The AtomicIntegerArray class describes an int array whose elements may be updated atomically.

19. The answer is false: volatile doesn’t support atomic read-modify-write sequences.

20. The Compare-and-Swap instruction is responsible for the performance gains offered by the Concurrency Utilities.

21. Listing A-47 presents the CountingThreads application that was called for in Chapter 10.
Listing A-47. Executor-Based Counting Threads

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                    System.out.println(name + ":
```

22. Listing A-48 presents the CountingThreads application with custom-named threads that was called for in Chapter 10.

Listing A-48. Executor-Based Counting Threads A and B

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadFactory;

public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                    System.out.println(name + ":
```
es.submit(r);
es = Executors.newSingleThreadExecutor(new NamedThread("B"));
es.submit(r);
}
}
class NamedThread implements ThreadFactory
{
    private String name;

    NamedThread(String name)
    {
        this.name = name;
    }

    @Override
    public Thread newThread(Runnable r)
    {
        return new Thread(r, name);
    }
}

23. Listing A-49 presents the DeadlockDemo application that was called for in Chapter 10.

Listing A-49. Demonstrating Deadlock via Lock and ReentrantLock

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DeadlockDemo
{
    private final Lock lock1 = new ReentrantLock();
    private final Lock lock2 = new ReentrantLock();

    public void m1()
    {
        lock1.lock();
        try
        {
            lock2.lock();
            try
            {
                System.out.println("first thread in m1()");
            }
            finally
            {
                lock2.unlock();
            }
        }
    }
finally
{
   lock1.unlock();
}

public void m2()
{
   lock2.lock();
   try
   {
      lock1.lock();
      try
      {
         System.out.println("second thread in m2()");
      }
      finally
      {
         lock1.unlock();
      }
   }
   finally
   {
      lock2.unlock();
   }
}

public static void main(String[] args)
{
   final DeadlockDemo dld = new DeadlockDemo();
   Runnable runnable1 = new Runnable()
   {
      @Override
      public void run()
      {
         while(true)
         {
            dld.m1();
            try
            {
               Thread.sleep(50);
            }
            catch (InterruptedException ie)
            {
               assert false;
            }
         }
      }
   };
}
Listing A-50 presents the DeadlockDemo application that avoids deadlock that was called for in Chapter 10.

Listing A-50. Demonstrating Deadlock Avoidance via Lock and ReentrantLock

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DeadlockDemo {
    private final Lock lock1 = new ReentrantLock();
    private final Lock lock2 = new ReentrantLock();

    public void m1() {
        if (lock1.tryLock())
            try {
                if (lock2.tryLock())
                    try {
                        System.out.println("first thread in m1()");
                    } catch (InterruptedException ie) {
                        assert false;
                    }
            } catch (InterruptedException ie) {
                assert false;
            }
    }

    ExecutorService executor1 = Executors.newSingleThreadExecutor();
    Runnable runnable1 = new Runnable() {
        @Override
        public void run() {
            while (true) {
                dld.m2();
                try {
                    Thread.sleep(50);
                } catch (InterruptedException ie) {
                    assert false;
                }
            }
        }
    };
    ExecutorService executor2 = Executors.newSingleThreadExecutor();
    executor1.submit(runnable1);
    executor2.submit(runnable2);
}
finally {
    lock2.unlock();
}
} finally {
    lock1.unlock();
}

public void m2()
{
    if (lock2.tryLock())
    try {
        if (lock1.tryLock())
        try {
            System.out.println("second thread in m2()");
        }
        finally {
            lock1.unlock();
        }
    }
    finally {
        lock2.unlock();
    }
}

public static void main(String[] args)
{
    final DeadlockDemo dld = new DeadlockDemo();
    Runnable runnable1 = new Runnable()
    {
        @Override
        public void run()
        {
            while(true)
            {
                dld.m1();
                try {
                    Thread.sleep(50);
                }
                catch (InterruptedException ie)
                {
                    assert false;
                }
            }
        }
    };
}
ExecutorService executor1 = Executors.newSingleThreadExecutor();
Runnable runnable2 = new Runnable()
{
    @Override
    public void run()
    {
        while(true)
        {
            dld.m2();
            try
            {
                Thread.sleep(50);
            }
            catch (InterruptedException ie)
            {
                assert false;
            }
        }
    }
};
ExecutorService executor2 = Executors.newSingleThreadExecutor();
executor1.submit(runnable1);
executor2.submit(runnable2);
}

24. The atomic variable equivalent of int total = ++counter; is as follows:

AtomicInteger counter = new AtomicInteger(0);
int total = counter.incrementAndGet();

The atomic variable equivalent of int total = counter--; is as follows:

AtomicInteger counter = new AtomicInteger(0);
int total = counter.getAndDecrement();

Chapter 11: Performing Classic I/O

1. The purpose of the File class is to offer access to the underlying platform’s available filesystem(s).

2. Instances of the File class contain the pathnames of files and directories that may or may not exist in their filesystems.

3. File’s listRoots() method returns an array of File objects denoting the root directories (roots) of available filesystems.

4. A path is a hierarchy of directories that must be traversed to locate a file or a directory. A pathname is a string representation of a path; a platform-dependent separator character (such as the Windows backslash \ character) appears between consecutive names.
5. The difference between an absolute pathname and a relative pathname is as follows: an absolute pathname is a pathname that starts with the root directory symbol, whereas a relative pathname is a pathname that doesn’t start with the root directory symbol (it’s interpreted via information taken from some other pathname).

6. You obtain the current user (also known as working) directory by specifying `System.getProperty("user.dir")`.

7. A parent pathname is a string that consists of all pathname components except for the last name.

8. Normalize means to replace separator characters with the default name-separator character so that the pathname is compliant with the underlying filesystem.

9. You obtain the default name-separator character by accessing `File`'s `separator` and `separatorChar` class fields. The first field stores the character as a `char` and the second field stores it as a `String`.

10. A canonical pathname is a pathname that’s absolute and unique, and it is formatted the same way every time.

11. The difference between `File`'s `getParent()` and `getName()` methods is that `getParent()` returns the parent pathname and `getName()` returns the last name in the pathname’s name sequence.

12. The answer is false: `File`'s `exists()` method determines whether or not a file or directory exists.

13. A normal file is a file that’s not a directory and that satisfies other platform-dependent criteria: it’s not a symbolic link or named pipe, for example. Any nondirectory file created by a Java application is guaranteed to be a normal file.

14. `File`'s `lastModified()` method returns the time that the file denoted by this `File` object’s pathname was last modified or 0 when the file doesn’t exist or an I/O error occurred during this method call. The returned value is measured in milliseconds since the Unix epoch (00:00:00 GMT, January 1, 1970).

15. The answer is true: `File`'s `list()` method returns an array of `String`s where each entry is a filename rather than a complete path.

16. The difference between the `FilenameFilter` and `FileFilter` interfaces is as follows: `FilenameFilter` declares a single boolean `accept(File dir, String name)` method, whereas `FileFilter` declares a single boolean `accept(String pathname)` method. Either method accomplishes the same task of accepting (by returning true) or rejecting (by returning false) the inclusion of the file or directory identified by the argument(s) in a directory listing.
17. The answer is false: File's `createNewFile()` method checks for file existence and creates the file when it doesn't exist in a single operation that's atomic with respect to all other filesystem activities that might affect the file.

18. The default temporary directory where File's `createTempFile(String, String)` method creates temporary files can be located by reading the `java.io.tmpdir` system property.

19. You ensure that a temporary file is removed when the virtual machine ends normally (it doesn’t crash and the power isn’t lost) by registering the temporary file for deletion through a call to File's `deleteOnExit()` method.

20. You would accurately compare two File objects by first calling File's `getCanonicalFile()` method on each File object and then comparing the returned File objects.

21. The purpose of the RandomAccessFile class is to create and/or open files for random access in which a mixture of write and read operations can occur until the file is closed.

22. The purpose of the "rwd" and "rws" mode arguments is to ensure that any writes to a file located on a local storage device are written to the device, which guarantees that critical data isn't lost when the system crashes. No guarantee is made when the file doesn't reside on a local device.

23. A file pointer is a cursor that identifies the location of the next byte to write or read. When an existing file is opened, the file pointer is set to its first byte at offset 0. The file pointer is also set to 0 when the file is created.

24. The answer is false: when you call RandomAccessFile's `seek(long)` method to set the file pointer's value, and when this value is greater than the length of the file, the file's length doesn't change. The file length will only change by writing after the offset has been set beyond the end of the file.

25. A flat file database is a single file organized into records and fields. A record stores a single entry (such as a part in a parts database) and a field stores a single attribute of the entry (such as a part number).

26. A stream is an ordered sequence of bytes of arbitrary length. Bytes flow over an output stream from an application to a destination, and flow over an input stream from a source to an application.

27. The purpose of OutputStream's `flush()` method is to write any buffered output bytes to the destination. If the intended destination of this output stream is an abstraction provided by the underlying platform (such as a file), flushing the stream only guarantees that bytes previously written to the stream are passed to the underlying platform for writing; it doesn’t guarantee that they're actually written to a physical device such as a disk drive.
28. The answer is true: `OutputStream`'s `close()` method automatically flushes the output stream. If an application ends before `close()` is called, the output stream is automatically closed and its data is flushed.

29. The purpose of `InputStream`'s `mark(int)` and `reset()` methods is to reread a portion of a stream. `mark(int)` marks the current position in this input stream. A subsequent call to `reset()` repositions this stream to the last marked position so that subsequent read operations reread the same bytes. Don’t forget to call `markSupported()` to find out if the subclass supports `mark()` and `reset()`.

30. You would access a copy of a `ByteArrayOutputStream` instance’s internal byte array by calling `ByteArrayOutputStream`'s `toByteArray()` method.

31. The answer is false: `FileOutputStream` and `FileInputStream` don’t provide internal buffers to improve the performance of write and read operations.

32. You would use `PipedOutputStream` and `PipedInputStream` to communicate data between a pair of executing threads.

33. A filter stream is a stream that buffers, compresses/uncompresses, encrypts/decrypts, or otherwise manipulates an input stream’s byte sequence before it reaches its destination.

34. Two streams are chained together when a stream instance is passed to another stream class’s constructor.

35. You improve the performance of a file output stream by chaining a `BufferedOutputStream` instance to a `FileOutputStream` instance and calling the `BufferedOutputStream` instance’s `write()` methods so that data is buffered before flowing to the file output stream. You improve the performance of a file input stream by chaining a `BufferedInputStream` instance to a `FileInputStream` instance so that data flowing from a file input stream is buffered before being returned from the `BufferedInputStream` instance by calling this instance’s `read()` methods.

36. `DataOutputStream` and `DataInputStream` support `FileOutputStream` and `FileInputStream` by providing methods to write and read primitive-type values and strings in a platform-independent way. In contrast, `FileOutputStream` and `FileInputStream` provide methods for writing/reading bytes and arrays of bytes only.

37. Object serialization is a virtual machine mechanism for serializing object state into a stream of bytes. Its deserialization counterpart is a virtual machine mechanism for deserializing this state from a byte stream.

38. The three forms of serialization and deserialization that Java supports are default serialization and deserialization, custom serialization and deserialization, and externalization.
39. The purpose of the `Serializable` interface is to tell the virtual machine that it's okay to serialize objects of the implementing class.

40. When the serialization mechanism encounters an object whose class doesn't implement `Serializable`, it throws an instance of the `NotSerializableException` class.

41. The three stated reasons for Java not supporting unlimited serialization are as follows: security, performance, and objects not amenable to serialization.

42. You initiate serialization by creating an `ObjectOutputStream` instance and calling its `writeObject()` method. You initiate deserialization by creating an `ObjectInputStream` instance and calling its `readObject()` method.

43. The answer is false: class fields are not automatically serialized.

44. The purpose of the `transient` reserved word is to mark instance fields that don't participate in default serialization and default deserialization.

45. The deserialization mechanism causes `readObject()` to throw an instance of the `InvalidClassException` class when it attempts to deserialize an object whose class has changed.

46. The deserialization mechanism detects that a serialized object's class has changed as follows: every serialized object has an identifier, and the deserialization mechanism compares the identifier of the object being deserialized with the serialized identifier of its class (all serializable classes are automatically given unique identifiers unless they explicitly specify their own identifiers) and causes `InvalidClassException` to be thrown when it detects a mismatch.

47. You can add an instance field to a class and avoid trouble when deserializing an object that was serialized before the instance field was added by introducing a `long serialVersionUID = long integer value;` declaration into the class. The `long integer value` must be unique, and it is known as a stream unique identifier (SUID). You can use the JDK's `serialver` tool to help with this task.

48. You customize the default serialization and deserialization mechanisms without using externalization by declaring private `void writeObject(ObjectOutputStream)` and `void readObject(ObjectInputStream)` methods in the class.

49. You tell the serialization and deserialization mechanisms to serialize or deserialize the object's normal state before serializing or deserializing additional data items by first calling `ObjectOutputStream's defaultWriteObject()` method in `writeObject(ObjectOutputStream)` and by first calling `ObjectInputStream's defaultReadObject()` method in `readObject(ObjectInputStream)`.

50. Externalization differs from default and custom serialization and deserialization in that it offers complete control over the serialization and deserialization tasks.
51. A class indicates that it supports externalization by implementing the Externalizable interface instead of Serializable and by declaring void writeExternal(ObjectOutput) and void readExternal(ObjectInput in) methods instead of void writeObject(ObjectOutputStream) and void readObject(ObjectInputStream) methods.

52. The answer is true: during externalization, the deserialization mechanism throws InvalidClassException with a “no valid constructor” message when it doesn't detect a public noargument constructor.

53. The difference between PrintStream's print() and println() methods is that the print() methods don’t append a line terminator to their output, whereas the println() methods append a line terminator.

54. PrintStream's noargument void println() method outputs the line. separator system property's value to ensure that lines are terminated in a portable manner (such as a carriage return followed by a newline/line feed on Windows, or only a newline/line feed on Unix/Linux).

55. Java's stream classes are not good at streaming characters because bytes and characters are two different things: a byte represents an 8-bit data item and a character represents a 16-bit data item. Also, byte streams have no knowledge of character sets and their character encodings.

56. Java provides writer and reader classes as the preferred alternative to stream classes when it comes to character I/O.

57. The answer is false: Reader doesn't declare an available() method.

58. The purpose of the OutputStreamWriter class is to serve as a bridge between an incoming sequence of characters and an outgoing stream of bytes. Characters written to this writer are encoded into bytes according to the default or specified character encoding. The purpose of the InputStreamReader class is to serve as a bridge between an incoming stream of bytes and an outgoing sequence of characters. Characters read from this reader are decoded from bytes according to the default or specified character encoding.

59. You identify the default character encoding by reading the value of the file.encoding system property.

60. The purpose of the FileWriter class is to connect conveniently to the underlying file output stream using the default character encoding. The purpose of the FileReader class is to connect conveniently to the underlying file input stream using the default character encoding.

61. Listing A-51 presents the Touch application that was called for in Chapter 11.
Listing A-51. Setting a File or Directory’s Timestamp to the Current Time

import java.io.File;
import java.util.Date;

public class Touch
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java Touch pathname");
            return;
        }
        new File(args[0]).setLastModified(new Date().getTime());
    }
}

62. Listing A-52 presents the Copy application that was called for in Chapter 11.

Listing A-52. Copying a Source File to a Destination File with Buffered I/O

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy
{
    public static void main(String[] args)
    {
        if (args.length != 2)
        {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        try
        {
            FileInputStream fis = new FileInputStream(args[0]);
            bis = new BufferedInputStream(fis);
            FileOutputStream fos = new FileOutputStream(args[1]);
            bos = new BufferedOutputStream(fos);
            int b; // I chose b instead of byte because byte is a reserved word.
            while ((b = bis.read()) != -1)
            {
                bos.write(b);
            }
        }
        catch (FileNotFoundException e)
        {
            System.err.println("FileNotFoundException: " + e.getMessage());
        }
        catch (IOException e)
        {
            System.err.println("IOException: " + e.getMessage());
        } finally
        {
            if (bis != null)
            {
                try
                {
                    bis.close();
                }
                catch (IOException e)
                {
                    System.err.println("IOException: " + e.getMessage());
                }
            }
            if (bos != null)
            {
                try
                {
                    bos.close();
                }
                catch (IOException e)
                {
                    System.err.println("IOException: " + e.getMessage());
                }
            }
        }
    }
}
catch (FileNotFoundException fnfe) {
    System.err.println(args[0] + " could not be opened for input, or " +
    args[1] + " could not be created for output");
}
catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
}
finally {
    if (bis != null)
        try {
            bis.close();
        } catch (IOException ioe) {
            assert false; // shouldn't happen in this context
        }
    if (bos != null)
        try {
            bos.close();
        } catch (IOException ioe) {
            assert false; // shouldn't happen in this context
        }
}

63. Listing A-53 presents the Split application that was called for in Chapter 11.

Listing A-53. Splitting a Large File into Numerous Smaller Part Files

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Split {
    static final int FILESIZE = 1400000;
    static byte[] buffer = new byte[FILESIZE];
public static void main(String[] args)
{
    if (args.length != 1)
    {
        System.err.println("usage: java Split pathname");
        return;
    }
    File file = new File(args[0]);
    long length = file.length();
    int nWholeParts = (int) (length / FILESIZE);
    int remainder = (int) (length % FILESIZE);
    System.out.printf("Splitting %s into %d parts\n", args[0],
        (remainder == 0) ? nWholeParts : nWholeParts + 1);
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;
    try
    {
        FileInputStream fis = new FileInputStream(args[0]);
        bis = new BufferedInputStream(fis);
        for (int i = 0; i < nWholeParts; i++)
        {
            bis.read(buffer);
            System.out.println("Writing part " + i);
            FileOutputStream fos = new FileOutputStream("part" + i);
            bos = new BufferedOutputStream(fos);
            bos.write(buffer);
            bos.close();
            bos = null;
        }
        if (remainder != 0)
        {
            int br = bis.read(buffer);
            if (br != remainder)
            {
                System.err.println("Last part mismatch: expected " + remainder
                    + " bytes");
                System.exit(0);
            }
            System.out.println("Writing part " + nWholeParts);
            FileOutputStream fos = new FileOutputStream("part" + nWholeParts);
            bos = new BufferedOutputStream(fos);
            bos.write(buffer, 0, remainder);
        }
    }
    catch (IOException ioe)
    {
        ioe.printStackTrace();
    }
}
1082 APPENDIX A: Solutions to Exercises

finally {
  if (bis != null)
    try {
      bis.close();
    } catch (IOException ioe) {
      assert false; // shouldn't happen in this context
    }
  if (bos != null)
    try {
      bos.close();
    } catch (IOException ioe) {
      assert false; // shouldn't happen in this context
    }
}

64. Listing A-54 presents the CircleInfo application that was called for in Chapter 11.

Listing A-54. Reading Lines of Text from Standard Input That Represent Circle Radii and Outputting Circumference and Area Based on the Current Radius

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class CircleInfo {
  public static void main(String[] args) throws IOException {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    while (true) {
      System.out.print("Enter circle's radius: ");
      String str = br.readLine();
      double radius;
      try {
        radius = Double.valueOf(str).doubleValue();
        if (radius <= 0)
          System.err.println("radius must not be 0 or negative");
        else
          System.out.println("Circumference: " + Math.PI * 2.0 * radius);
      }
    }
  }
}
Chapter 12: Accessing Networks

1. A network is a group of interconnected nodes that can be shared among the network’s users.

2. An intranet is a network located within an organization, and an internet is a network connecting organizations to each other.

3. Intranets and internets often use TCP/IP to communicate between nodes. Transmission Control Protocol (TCP) is a connection-oriented protocol, User Datagram Protocol (UDP) is a connectionless protocol, and Internet Protocol (IP) is the basic protocol over which TCP and UDP perform their tasks.

4. A host is a computer-based TCP/IP node.

5. A socket is an endpoint in a communications link between two processes.

6. A socket is identified by an IP address that identifies the host and by a port number that identifies the process running on that host.

7. An IP address is a 32-bit or 128-bit unsigned integer that uniquely identifies a network host or some other network node.

8. A packet is an addressable message chunk. Packets are often referred to as IP datagrams.

9. A socket address is comprised of an IP address and a port number.

10. The InetAddress subclasses that are used to represent IPv4 and IPv6 addresses are Inet4Address and Inet6Address.

11. The loopback interface is a software-based network interface where outgoing data loops back as incoming data.

12. The answer is false: in network byte order, the most significant byte comes first.

13. The local host is represented by hostname localhost or by an IP address that’s commonly expressed as 127.0.0.1 (IPv4) or ::1 (IPv6).

14. A socket option is a parameter for configuring socket behavior.
15. Socket options are described by constants that are declared in the 
   SocketOptions interface.

16. The answer is false: you don’t set a socket option by calling the void
   setOption(int optID, Object value) method. Instead, you call one of the 
   type-safe socket option methods that are declared in a Socket-suffixed class.

17. Sockets based on the Socket class are commonly referred to as stream 
   sockets because Socket is associated with the InputStream and 
   OutputStream classes.

18. In the context of a Socket instance, binding makes a client socket address 
   available to a server socket so that a server process can communicate with 
   the client process via the server socket.

19. A proxy is a host that sits between an intranet and the Internet for security 
   purposes. Java represents proxy settings via instances of the 
   java.net.Proxy class.

20. The answer is false: the ServerSocket() constructor creates an unbound 
   server socket.

21. The difference between the DatagramSocket and MulticastSocket classes is
   as follows: DatagramSocket lets you perform UDP-based communications 
   between a pair of hosts, whereas MulticastSocket lets you perform UDP-
   based communications between many hosts.

22. A datagram packet is an array of bytes associated with an instance of the 
   DatagramPacket class.

23. The difference between unicasting and multicasting is as follows: unicasting 
   is the act of a server sending a message to a single client, whereas 
   multicasting is the act of a server sending a message to multiple clients.

24. A URL is a character string that specifies where a resource (such as a web 
   page) is located on a TCP/IP-based network (such as the Internet). Also, it 
   provides the means to retrieve that resource.

25. A URN is a character string that names a resource and doesn’t provide a way 
   to access that resource (the resource might not be available).

26. The answer is true: URLs and URNs are also URIs.

27. The URL(String s) constructor throws MalformedURLException when you 
   pass null to s.

28. The equivalent of openStream() is to execute openConnection().getInputStream().

29. The answer is false: you don’t need to invoke URLConnection’s void 
   setDoInput(boolean doInput) method with true as the argument before you 
   can input content from a web resource. The default setting is true.
30. When it encounters a space character, URLEncoder converts it to a plus sign.

31. The purpose of the URI class is to represent names (URNs) and resources (URLs). Also, it provides normalization, resolution, and relativization operations; the resulting URI can be converted into a URL as long as it represents a resource.

32. Normalization is the process of removing unnecessary “.” and “..” path segments from a hierarchical URI’s path component. Each “.” segment is removed. A “..” segment is removed only when it’s preceded by a non-“..” segment.

33. The answer is true: resolution and relativization are inverse operations of each other.

34. The NetworkInterface class represents a network interface as a name and a list of IP addresses assigned to this interface. Furthermore, it’s used to identify the local interface on which a multicast group is joined.

35. A MAC address is an array of bytes containing a network interface’s hardware address.

36. MTU stands for Maximum Transmission Unit. This size represents the maximum length of a message that can fit into an IP datagram without needing to fragment the message into multiple IP datagrams.

37. The answer is false: NetworkInterface’s getName() method returns a network interface’s name (such as eth0 or lo), not a human-readable display name.

38. InterfaceAddress’s getNetworkPrefixLength() method returns the subnet mask under IPv4.

39. HTTP cookie (cookie for short) is a state object.

40. It’s preferable to store cookies on the client rather than on the server because of the potential for millions of cookies (depending on a web site’s popularity).

41. The four java.net types that are used to work with cookies are CookieHandler, CookieManager, CookiePolicy, and CookieStore.

42. Listing A-55 presents the enhanced EchoClient application that was called for in Chapter 12.

Listing A-55. Echoing Data to and Receiving It Back from a Server and Explicitly Closing the Socket

```java
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
```

www.it-ebooks.info
import java.net.Socket;
import java.net.UnknownHostException;

public class EchoClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage : java EchoClient message");
            System.err.println("example: java EchoClient 'This is a test.'");
            return;
        }
        Socket socket = null;
        try {
            socket = new Socket("localhost", 9999);
            OutputStream os = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(os);
            PrintWriter pw = new PrintWriter(osw);
            pw.println(args[0]);
            pw.flush();
            InputStream is = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            System.out.println(br.readLine());
        } catch (UnknownHostException uhe) {
            System.err.println("unknown host: "+ uhe.getMessage());
        } catch (IOException ioe) {
            System.err.println("I/O error: "+ ioe.getMessage());
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException ioe) {
                    assert false; // shouldn't happen in this context
                }
            }
        }
    }
}
43. Listing A-56 presents the enhanced EchoServer application that was called for in Chapter 12.

Listing A-56. Receiving Data from and Echoing It Back to a Client and Explicitly Closing the Socket After a kill File Appears

import java.io.BufferedReader;
import java.io.File;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    public static void main(String[] args) {
        System.out.println("Starting echo server..." KlUT); ServerSocket ss = null;
        try {
            ss = new ServerSocket(9999);
            File file = new File("kill");
            while (!file.exists())
                try { Socket s = ss.accept(); // waiting for client request
                    try {
                        InputStream is = s.getInputStream();
                        InputStreamReader isr = new InputStreamReader(is);
                        BufferedReader br = new BufferedReader(isr);
                        String msg = br.readLine();
                        System.out.println(msg);
                        OutputStream os = s.getOutputStream();
                        OutputStreamWriter osw = new OutputStreamWriter(os);
                        PrintWriter pw = new PrintWriter(osw);
                        pw.println(msg);
                        pw.flush();
                    } catch (IOException ioe) {
                        System.err.println("I/O error: " + ioe.getMessage());
                    }
                }
            
        }
    }
}
finally
{
    try
    {
        s.close();
    }
    catch (IOException ioe)
    {
        assert false; // shouldn't happen in this context
    }
}
}

Chapter 13: Migrating to New I/O

1. New I/O is a more powerful I/O architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture largely consists of buffers, channels, selectors, regular expressions, and charsets, but it also could be considered to include a printf-style formatting facility.

2. A buffer is an object that stores a fixed amount of data to be sent to or received from an I/O service (a means for performing input/output). It sits between an application and a channel that writes the buffered data to the service or reads the data from the service and deposits it into the buffer.

3. A buffer's four properties are capacity, limit, position, and mark.

4. When you invoke Buffer's array() method on a buffer backed by a read-only array, this method throws ReadOnlyBufferException.
5. When you invoke Buffer’s flip() method on a buffer, the limit is set to the current position and then the position is set to zero. When the mark is defined, it’s discarded. The buffer is now ready to be drained.

6. When you invoke Buffer’s reset() method on a buffer where a mark has not been set, this method throws InvalidMarkException.

7. The answer is false: buffers are not thread-safe.

8. The classes that extend the abstract Buffer class are ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer. Furthermore, this package includes MappedByteBuffer as an abstract ByteBuffer subclass.

9. You create a byte buffer by invoking one of its allocate(), allocateDirect(), or wrap() class methods.

10. A view buffer is a buffer that manages another buffer’s data.

11. A view buffer is created by calling a Buffer subclass’s duplicate() method.

12. You create a read-only view buffer by calling a Buffer subclass method such as ByteBuffer asReadOnlyBuffer() or CharBuffer asReadOnlyBuffer().

13. ByteBuffer’s methods for storing a single byte in a byte buffer are ByteBuffer put(int index, byte b) and ByteBuffer put(byte b). ByteBuffer’s methods for fetching a single byte from a byte buffer are byte get(int index) and byte get().

14. Attempting to use the relative put() method or the relative get() method when the current position is greater than or equal to the limit causes BufferOverflowException or BufferUnderflowException to occur.

15. The equivalent of executing buffer.flip(); is to execute buffer.limit(buffer.position()).position(0);

16. The answer is false: calling flip() twice doesn’t return you to the original state. Instead, the buffer has a zero size.

17. The difference between Buffer’s clear() and reset() methods is as follows: the clear() method marks a buffer as empty, whereas reset() changes the buffer’s current position to the previously set mark or throws InvalidMarkException when there’s no previously set mark.

18. ByteBuffer’s compact() method compacts a buffer by copying all bytes between the current position and the limit to the beginning of the buffer. The byte at index \( p = \text{position}() \) is copied to index 0, the byte at index \( p + 1 \) is copied to index 1, and so on until the byte at index \( \text{limit}() - 1 \) is copied to index \( n = \text{limit}() - 1 - p \). The buffer’s current position is then set to \( n + 1 \) and its limit is set to its capacity. The mark, when defined, is discarded.
19. The purpose of the `ByteOrder` class is to help you deal with byte-order issues when writing/reading multibyte values to/from a multibyte buffer.

20. A direct byte buffer is a byte buffer that interacts with channels and native code to perform I/O. The direct byte buffer attempts to store byte elements in a memory area that a channel uses to perform direct (raw) access via native code that tells the operating system to drain or fill the memory area directly.

21. You obtain a direct byte buffer by invoking `ByteBuffer`'s `allocateDirect()` method.

22. A channel is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that's capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations.

23. The capabilities that the `Channel` interface provides are closing a channel (via the `close()` method) and determining whether or not a channel is open (via the `isOpen()` method.)

24. The three interfaces that directly extend `Channel` are `WritableByteChannel`, `ReadableByteChannel`, and `InterruptibleChannel`.

25. The answer is true: a channel that implements `InterruptibleChannel` is asynchronously closeable.

26. The two ways to obtain a channel are to invoke a `Channels` class method, such as `WritableByteChannel newChannel(OutputStream outputStream)`, and to invoke a channel method on a classic I/O class, such as `RandomAccessFile`'s `FileChannel getChannel()` method.

27. Scatter/gather I/O is the ability to perform a single I/O operation across multiple buffers.

28. The `ScatteringByteChannel` and `GatheringByteChannel` interfaces are provided for achieving scatter/gather I/O.

29. A file channel is a channel to an underlying file.

30. The answer is false: file channels support scatter/gather I/O.

31. An exclusive lock is a lock that prevents other file locks from being used within the region governed by the exclusive lock. In contrast, a shared lock is a lock that may apply to a region governed by other shared locks.

32. The fundamental difference between `FileChannel`'s `lock()` and `tryLock()` methods is that the `lock()` methods can block and the `tryLock()` methods never block.
33. The FileLock lock() method throws OverlappingFileLockException when either a lock is already held that overlaps this lock request or another thread is waiting to acquire a lock that will overlap with this request.

34. The pattern that you should adopt to ensure that an acquired file lock is always released follows:

```java
FileLock lock = fileChannel.lock();
try {
    // interact with the file channel
}
catch (IOException ioe) {
    // handle the exception
}
finally {
    lock.release();
}
```

35. FileChannel provides the MappedByteBuffer map(FileChannel.MapMode mode, long position, long size) method for mapping a region of a file into memory.

36. The three file-mapping modes are read-only, read-write, and private. They’re described by the READ_ONLY, READ_WRITE, and PRIVATE constants declared by the FileChannel.MapMode enumerated type.

37. The private file-mapping mode corresponds to copy-on-write. Changes made to the resulting buffer will not be propagated to the file and will not be visible to other programs that have mapped the same file. Instead, changes will cause private copies of the modified portions of the buffer to be created. These changes are lost when the buffer is garbage collected.

38. The FileChannel methods that optimize the common practice of performing bulk transfers are transferFrom() and transferTo().

39. The answer is true: socket channels are selectable and can function in nonblocking mode.

40. The three classes that describe socket channels are ServerSocketChannel, SocketChannel, and DatagramChannel.

41. The answer is false: datagram channels are thread-safe.

42. Socket channels support nonblocking mode because the blocking nature of sockets created from Java's socket classes is a serious limitation to a network-oriented Java application's scalability.

43. You would obtain a socket channel's associated socket by invoking its socket() method.
44. You obtain a server socket channel by invoking ServerSocketChannel’s open() class method.

45. A selector is an object created from a subclass of the abstract Selector class. It maintains a set of channels, which it examines to determine which of them are ready for reading, writing, completing a connection sequence, accepting another connection, or some combination of these tasks. The actual work is delegated to the operating system via a POSIX select() or similar system call.

46. The three main types that support selectors are SelectableChannel, SelectionKey, and Selector.

47. The answer is false: file channels cannot be used with selectors. Only channels that implement SelectableChannel can be used with selectors. FileChannel doesn’t implement SelectableChannel.

48. A regular expression (also known as a regex or regexp) is a string-based pattern that represents the set of strings that match this pattern.

49. Instances of the Pattern class represent patterns via compiled regexes. Regexes are compiled for performance reasons; pattern matching via compiled regexes is much faster than if the regexes were not compiled.

50. Pattern’s compile() methods throw PatternSyntaxException when they discover illegal syntax in their regular expression arguments.

51. Instances of the Matcher class attempt to match compiled regexes against input text.

52. The difference between Matcher’s matches() and lookingAt() methods is that unlike matches(), lookingAt() doesn’t require the entire region to be matched.

53. A character class is a set of characters appearing between [ and ].

54. There are six kinds of character classes: simple, negation, range, union, intersection, and subtraction.

55. A capturing group saves a match’s characters for later recall during pattern matching.

56. A zero-length match is a match of zero length in which the start and end indexes are equal.

57. A quantifier is a numeric value implicitly or explicitly bound to a pattern. Quantifiers are categorized as greedy, reluctant, or possessive.

58. The difference between a greedy quantifier and a reluctant quantifier is that a greedy quantifier attempts to find the longest match, whereas a reluctant quantifier attempts to find the shortest match.
59. Possessive and greedy quantifiers differ in that a possessive quantifier only makes one attempt to find the longest match, whereas a greedy quantifier can make multiple attempts.

60. The two main classes that contribute to the NIO printf-style formatting facility are Formatter and Scanner.

61. The %n format specifier outputs a platform-specific line separator.

62. Listing A-57 presents the enhanced Copy application that was called for in Chapter 13.

**Listing A-57. Copying a File via a Byte Buffer and a File Channel**

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        FileChannel fcSrc = null;
        FileChannel fcDest = null;
        try {
            FileInputStream fis = new FileInputStream(args[0]);
            fcSrc = fis.getChannel();
            FileOutputStream fos = new FileOutputStream(args[1]);
            fcDest = fos.getChannel();
            ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
            while ((fcSrc.read(buffer)) != -1) {
                buffer.flip();
                while (buffer.hasRemaining()) {
                    fcDest.write(buffer);
                    buffer.clear();
                }
            }
        }
    }
}
```
catch (FileNotFoundException fnfe)
{
    System.err.println(args[0] + " could not be opened for input, or " +
                     args[1] + " could not be created for output");
}
catch (IOException ioe)
{
    System.err.println("I/O error: " + ioe.getMessage());
}
finally
{
    if (fcSrc != null)
    try
    {
        fcSrc.close();
    }
    catch (IOException ioe)
    {
        assert false; // shouldn't happen in this context
    }

    if (fcDest != null)
    try
    {
        fcDest.close();
    }
    catch (IOException ioe)
    {
        assert false; // shouldn't happen in this context
    }
}

63. Listing A-58 presents the ReplaceText application that was called for in
Chapter 13.

Listing A-58. Replacing All Matches of the Pattern with Replacement Text

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class ReplaceText
{
    public static void main(String[] args)
    {
        if (args.length != 3)
        {
            System.err.println("usage: java ReplaceText text oldText newText");
            return;
        }
try {
    Pattern p = Pattern.compile(args[1]);
    Matcher m = p.matcher(args[0]);
    String result = m.replaceAll(args[2]);
    System.out.println(result);
} catch (PatternSyntaxException pse) {
    System.err.println(pse);
}

Listing A-59 presents the ValidateInput application that was called for in Chapter 13.

Listing A-59. Using a Scanner to Validate That Each Input Line Contains a String Name Followed by an Integer Age

```java
import java.util.Scanner;

public class ValidateInput {
    public static void main(String[] args) {
        // Scan standard input.
        Scanner scannerInput = new Scanner(System.in);

        // Keep track of current line number.
        int lineNo = 0;

        // Scan this input on a line-by-line basis.
        while (scannerInput.hasNextLine()) {
            // Obtain current line.
            String curLine = scannerInput.nextLine();

            // Output line.
            System.out.printf("%d: %s%n", ++lineNo, curLine);

            // Obtain a scanner to scan the current line.
            Scanner scannerLine = new Scanner(curLine);

            // Verify that the line has a name field.
            if (scannerLine.hasNext()) {
                System.out.printf("Name: %s%n", scannerLine.next());
            } else {
                System.out.printf("%d: name field missing%n", lineNo);
                continue;
            }
        }
    }
```
// Verify that the line has a second age field, of type integer.
if (scannerLine.hasNextInt())
    System.out.printf("Age: %d\n", scannerLine.nextInt());
else
{
    System.out.printf("%d: age field missing\n\n", lineNo);
    continue;
}

System.out.println();

// Close current line scanner.
scannerLine.close();

// Close standard input scanner.
scannerInput.close();

Chapter 14: Accessing Databases

1. A database is an organized collection of data.

2. A relational database is a database that organizes data into tables that can be related to each other.

3. Two other database categories are hierarchical databases and object-oriented databases.

4. A database management system is a set of programs that enables you to store, modify, and extract information from a database. It also provides users with tools to add, delete, access, modify, and analyze data stored in one location.

5. Java DB is a distribution of Apache’s open-source Derby product, which is based on IBM’s Cloudscape RDBMS code base.

6. The answer is false: Java DB’s embedded driver causes the database engine to run in the same virtual machine as the application.

7. setEmbeddedCP adds derby.jar and derbytools.jar to the classpath so that you can access Java DB’s embedded driver from your application.

8. The answer is false: you run Java DB’s sysinfo command-line tool to view the Java environment/Java DB configuration.

9. SQLite is a very simple and popular RDBMS that implements a self-contained, serverless, zero-configuration, transactional SQL database engine, and it is considered to be the most widely deployed database engine in the world.
10. Manifest typing is the ability to store any value of any data type into any column regardless of the declared type of that column.

11. SQLite provides the sqlite3 tool for accessing and modifying SQLite databases.

12. JDBC is an API for communicating with RDBMSes in an RDBMS-independent manner.

13. A data source is a data-storage facility ranging from a simple file to a complex relational database managed by an RDBMS.


15. The answer is false: there are four kinds of JDBC drivers.

16. A type three JDBC driver doesn’t depend on native code and communicates with a middleware server via an RDBMS-independent protocol. The middleware server then communicates the client’s requests to the data source.

17. JDBC provides the java.sql.DriverManager class and the javax.sql.DataSource interface for communicating with a data source.

18. You obtain a connection to a Java DB data source via the embedded driver by passing a URL of the form jdbc:derby:databaseName; URLAttributes to one of DriverManager’s getConnection() methods.

19. The answer is false: int getErrorCode() returns a vendor-specific error code.

20. A SQL state error code is a five-character string consisting of a two-character class value followed by a three-character subclass value.

21. The difference between SQLNonTransientException and SQLTransientException is as follows: SQLNonTransientException describes failed operations that cannot be retried without changing application source code or some aspect of the data source, and SQLTransientException describes failed operations that can be retried immediately.

22. JDBC’s three statement types are Statement, PreparedStatement, and CallableStatement.

23. The Statement method that you call to execute an SQL SELECT statement is ResultSet executeQuery(String sql).

24. A result set’s cursor provides access to a specific row of data.

25. The SQL FLOAT type maps to Java’s double type.

26. A prepared statement represents a precompiled SQL statement.

27. The answer is true: CallableStatement extends PreparedStatement.

28. A stored procedure is a list of SQL statements that perform a specific task.
29. You call a stored procedure by first obtaining a CallableStatement implementation instance (via one of Connection’s prepareCall() methods) that's associated with an escape clause, by next executing CallableStatement methods such as void setInt(String parameterName, int x) to pass arguments to escape clause parameters, and by finally invoking the boolean execute() method that CallableStatement inherits from its PreparedStatement superinterface.

30. An escape clause is RDBMS-independent syntax.

31. Metadata is data about data.

32. Metadata includes a list of catalogs, base tables, views, indexes, schemas, and additional information.

33. Listing A-60 presents the enhanced JDBCDemo application that was called for in Chapter 14.

Listing A-60. Outputting Database Metadata for the SQLite or JavaDB Embedded Driver

```java
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo {
    final static String URL1 = "jdbc:derby:employee;create=true";
    final static String URL2 = "jdbc:sqlite:employee";

    public static void main(String[] args) {
        String url = null;
        if (args.length != 1) {
            System.err.println("usage 1: java JDBCDemo javadb");
            System.err.println("usage 2: java JDBCDemo sqlite");
            return;
        }
        if (args[0].equals("javadb"))
            url = URL1;
        else
            if (args[0].equals("sqlite"))
                url = URL2;
            else {
                System.err.println("invalid command-line argument");
                return;
            }
```
Connection con = null;
try {
    if (args[0].equals("sqlite"))
        Class.forName("org.sqlite.JDBC");
    con = DriverManager.getConnection(url);
    dump(con.getMetaData());
} catch (ClassNotFoundException cnfe) {
    System.err.println("unable to load sqlite driver");
} catch (SQLException sqlex) {
    while (sqlex != null) {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally {
    if (con != null) {
        try {
            con.close();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }
}

static void dump(DatabaseMetaData dbmd) throws SQLException {
    System.out.println("DB Major Version = " + dbmd.getDatabaseMajorVersion());
    System.out.println("DB Minor Version = " + dbmd.getDatabaseMinorVersion());
    System.out.println("DB Product = " + dbmd.getDatabaseProductName());
    System.out.println("Driver Name = " + dbmd.getDriverName());
    System.out.println("Numeric function names for escape clause = " +
            dbmd.getNumericFunctions());
    System.out.println("String function names for escape clause = " +
            dbmd.getStringFunctions());
    System.out.println("System function names for escape clause = " +
            dbmd.getSystemFunctions());
    System.out.println("Time/date function names for escape clause = " +
            dbmd.getTimeDateFunctions());
    System.out.println("Catalog term: " + dbmd.getCatalogTerm());
}
System.out.println("Schema term: "+ dbmd.getSchemaTerm());
System.out.println();
System.out.println("Catalogs");
System.out.println("--------");
ResultSet rsCat = dbmd.getCatalogs();
while (rsCat.next())
    System.out.println(rsCat.getString("TABLE_CAT"));
System.out.println();
System.out.println("Schemas");
System.out.println("-------");
ResultSet rsSchem = dbmd.getSchemas();
while (rsSchem.next())
    System.out.println(rsSchem.getString("TABLE_SCHEM"));
System.out.println();
System.out.println("Schema/Table");
System.out.println("------------");
rsSchem = dbmd.getSchemas();
while (rsSchem.next())
    {
        String schem = rsSchem.getString("TABLE_SCHEM");
        ResultSet rsTab = dbmd.getTables(null, schem, "%", null);
        while (rsTab.next())
            System.out.println(schem + " " + rsTab.getString("TABLE_NAME"));
    }

Chapter 15: Parsing, Creating, and Transforming XML Documents

1. XML (eXtensible Markup Language) is a metalanguage for defining vocabularies (custom markup languages), which is key to XML's importance and popularity.

2. The answer is true: XML and HTML are descendents of SGML.

3. The XML declaration is special markup that informs an XML parser that the document is XML.

4. The XML declaration's three attributes are version, encoding, and standalone. The version attribute is nonoptional.

5. The answer is false: an element can consist of the empty-element tag, which is a standalone tag whose name ends with a forward slash (/), such as <break/>.

6. Following the XML declaration, an XML document is anchored in a root element.
7. Mixed content is a combination of child elements and content.

8. A character reference is a code that represents the character. The two kinds of character references are numeric character references (such as `&#x03A3;`) and character entity references (such as `&lt;`).

9. A CDATA section is a section of literal HTML or XML markup and content surrounded by the `<![CDATA[ prefix and the ]]>` suffix. You would use a CDATA section when you have a large amount of HTML/XML text and don’t want to replace each literal `< (start of tag) and `&` (start of entity) character with its `&lt;` and `&amp;` predefined character entity reference, which is a tedious and possibly error prone undertaking—you might forget to replace one of these characters.

10. A namespace is a Uniform Resource Identifier-based container that helps differentiate XML vocabularies by providing a unique context for its contained identifiers.

11. A namespace prefix is an alias for the URI.

12. The answer is true: a tag’s attributes don’t need to be prefixed when those attributes belong to the element.

13. A comment is a character sequence beginning with `<!--` and ending with `-->`. A comment can appear anywhere in an XML document except before the XML declaration, except within tags, and except within another comment.

14. A processing instruction is an instruction that’s made available to the application parsing the document. The instruction begins with `<?` and ends with `?>`.

15. The rules that an XML document must follow to be considered well formed are as follows: all elements must either have start and end tags or consist of empty-element tags, tags must be nested correctly, all attribute values must be quoted, empty elements must be properly formatted, and you must be careful with case. Furthermore, XML parsers that are aware of namespaces enforce two additional rules: all element and attribute names must not include more than one colon character; and no entity names, processing instruction targets, or notation names can contain colons.

16. For an XML document to be valid, the document must adhere to certain constraints. For example, one constraint might be that a specific element must always follow another specific element.

17. The two commonly used grammar languages are Document Type Definition and XML Schema.

18. The general syntax for declaring an element in a DTD is `<!ELEMENT name content-specifier>`.
19. XML Schema lets you create complex types from simple types.

20. SAX is an event-based API for parsing an XML document sequentially from start to finish. As a SAX-oriented parser encounters an item from the document’s infoset, it makes this item available to an application as an event by calling one of the methods in one of the application’s handlers, which the application has previously registered with the parser. The application can then consume this event by processing the infoset item in some manner.

21. You obtain a SAX 2-based parser by calling one of the org.xml.sax.helpers.XMLReaderFactory class’s createXMLReader() methods, which returns an XMLReader instance.

22. The purpose of the XMLReader interface is to describe a SAX parser. This interface makes available several methods for configuring the SAX parser and parsing an XML document’s content.

23. You tell a SAX parser to perform validation by invoking XMLReader’s setFeature(String name, boolean value) method, passing "http://xml.org/sax/features/validation" to name and true to value.

24. The four kinds of SAX-oriented exceptions that can be thrown when working with SAX are SAXException, SAXNotRecognizedException, SAXNotSupportedException, and SAXParseException.

25. The interface that a handler class implements to respond to content-oriented events is org.xml.sax.ContentHandler.

26. The three other core interfaces that a handler class is likely to implement are org.xml.sax.DTDHandler, org.xml.sax.EntityResolver, and org.xml.sax.ErrorHandler.

27. Ignorable whitespace is whitespace located between tags where the DTD doesn’t allow mixed content.

28. The answer is false: void error(SAXParseException exception) is called only for recoverable errors.

29. The purpose of the org.xml.sax.helpers.DefaultHandler class is to serve as a convenience base class for SAX2 applications. It provides default implementations for all of the callbacks in the four core SAX2 handler interfaces: ContentHandler, DTDHandler, EntityResolver, and ErrorHandler.

30. An entity is aliased data. An entity resolver is an object that uses the public identifier to choose a different system identifier. Upon encountering an external entity, the parser calls the custom entity resolver to obtain this identifier.
31. DOM is an API for parsing an XML document into an in-memory tree of nodes and for creating an XML document from a tree of nodes. After a DOM parser has created a document tree, an application uses the DOM API to navigate over and extract infoset items from the tree’s nodes.

32. The answer is false: Java 7 and newer versions of Android support DOM Levels 1, 2, and 3.

33. The 12 different DOM nodes are attribute node, CDATA section node, comment node, document node, document fragment node, document type node, element node, entity node, entity reference node, notation node, processing instruction node, and text node.

34. You obtain a document builder by first instantiating DocumentBuilderFactory via one of its newInstance() methods and then invoking newDocumentBuilder() on the returned DocumentBuilderFactory instance to obtain a DocumentBuilder instance.

35. You use a document builder to parse an XML document by invoking one of DocumentBuilder’s parse() methods.

36. The answer is true: Document and all other org.w3c.dom interfaces that describe different kinds of nodes are subinterfaces of the Node interface.


38. When creating a new XML document, you cannot use the DOM API to specify the XML declaration’s encoding attribute.

39. A push parser is a parser that pushes parsing events to an application. The application provides a handler that responds to these events. The parser invokes the handler’s callback methods to execute application-specific code as XML constructs are detected. A pull parser is a parser that lets an application pull parsed XML constructs, one at a time, from the parser when these constructs are needed. Unlike a push parser, which drives the application, a pull parser is driven by the application.

40. The answer is false: Android uses XMLPULL V1 as its pull parser.

41. You obtain the pull parser by working with the org.xmlpull.v1 package’s XmlPullParserFactory and XmlPullParser types. First you invoke XmlPullParserFactory’s XmlPullParserFactory newInstance() class method to create and return a new XmlPullParserFactory instance for creating XML pull parsers. Next, you configure the factory instance, for example, whether or not the pull parser should be aware of namespaces. Finally, you invoke XmlPullParserFactory’s XmlPullParser newPullParser() method to create and return a new XmlPullParser instance using the currently configured factory parameters.
42. You use the pull parser to parse an XML document as follows: call XmlPullParser's getEventType() method to obtain the initial event type, and then use a while loop to repeatedly pull parser events while the current event type isn't equal to XmlPullParser.END_DOCUMENT. Each loop iteration first processes the start document, start tag, text, or end tag event type; and then invokes XmlPullParser's next() method to obtain the next event type.

43. XPath is a non-XML declarative query language (defined by the W3C) for selecting an XML document's infoset items as one or more nodes.

44. XPath is commonly used to simplify access to a DOM tree's nodes and, in the context of XSLT, to select those input document elements (via XPath expressions) that are to be copied to an output document.

45. The seven kinds of nodes that XPath recognizes are element, attribute, text, namespace, processing instruction, comment, and document.

46. The answer is false: XPath doesn’t recognize CDATA sections.

47. XPath provides location path expressions for selecting nodes. A location path expression locates nodes via a sequence of steps starting from the context node, which is the root node or some other document node that is the current node. The returned set of nodes might be empty, or it might contain one or more nodes.

48. The answer is true: in a location path expression, you must prefix an attribute name with the @ symbol.

49. The functions that XPath provides for selecting comment, text, and processing-instruction nodes are comment(), text(), and processing-instruction().

50. XPath provides wildcards for selecting unknown nodes. The * wildcard matches any element node regardless of the node's type. It doesn't match attributes, text nodes, comments, or processing-instruction nodes. When you place a namespace prefix before the *, only elements belonging to that namespace are matched. The node() wildcard is a function that matches all nodes. Finally, the @* wildcard matches all attribute nodes.

51. You perform multiple selections by using the vertical bar (|). For example, author/*|publisher/* selects the children of author and the children of publisher.

52. A predicate is a square bracket-delimited Boolean expression that's tested against each selected node. If the expression evaluates to true, that node is included in the set of nodes returned by the XPath expression; otherwise, the node isn’t included in the set.
53. The functions that XPath provides for working with nodesets are last(), position(), id(), local-name(), namespace-uri(), and name().

54. The three advanced features that XPath provides to overcome limitations with the XPath 1.0 language are namespace contexts, extension functions and function resolvers, and variables and variable resolvers.

55. XSLT is a family of languages for transforming and formatting XML documents.

56. XSLT accomplishes its work by using XSLT processors and stylesheets. An XSLT processor is a software component that applies an XSLT stylesheet (an XML-based template consisting of content and transformation instructions) to an input document (without modifying the document), and copies the transformed result to a result tree, which can be output to a file or output stream, or even piped into another XSLT processor for additional transformations.

57. Listing A-61 presents the books.xml document file that was called for in Chapter 15.

Listing A-61. A Document of Books

```xml
<?xml version="1.0"?>
<books>
  <book isbn="0201548550" pubyear="1992">
    <title>
      Advanced C++
    </title>
    <author>
      James O. Coplien
    </author>
    <publisher>
      Addison Wesley
    </publisher>
  </book>
  <book isbn="9781430210450" pubyear="2008">
    <title>
      Beginning Groovy and Grails
    </title>
    <author>
      Christopher M. Judd
    </author>
    <author>
      Joseph Faisal Nusairat
    </author>
    <author>
      James Shingler
    </author>
  </book>
</books>
```
Listing A-62 presents the enhanced books.xml document file with an internal DTD that was called for in Chapter 15.


```xml
<?xml version="1.0"?>
<!DOCTYPE books [
  <!ELEMENT books (book+)>
  <!ELEMENT book (title, author+, publisher)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT publisher (#PCDATA)>
  <!ATTLIST book isbn CDATA #REQUIRED>
  <!ATTLIST book pubyear CDATA #REQUIRED>
]>
<books>
  <book isbn="0201548550" pubyear="1992">
    <title>
      Advanced C++
    </title>
    <author>
      James O. Coplien
    </author>
    <publisher>
      Addison Wesley
    </publisher>
  </book>
  <book isbn="9781430210450" pubyear="2008">
    <title>
      Beginning Groovy and Grails
    </title>
    <author>
      Christopher M. Judd
    </author>
  </book>
</books>
```
Listing A-63 and Listing A-64 present the SAXSearch and Handler classes that were called for in Chapter 15.

**Listing A-63. A SAX Driver Class for Searching books.xml for a Specific Publisher's Books**

```java
import java.io.FileReader;
import java.io.IOException;

import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;

import org.xml.sax.helpers.XMLReaderFactory;

public class SAXSearch {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java SAXSearch publisher");
            return;
        }

        try {
            XMLReader xmlr = XMLReaderFactory.createXMLReader();
            Handler handler = new Handler(args[0]);
            xmlr.setContentHandler(handler);
        }
    }
}
```
Listing A-64. A SAX Callback Class Whose Methods Are Called by the SAX Parser

```java
import org.xml.sax.Attributes;
import org.xml.sax.SAXParseException;
import org.xml.sax.ext.DefaultHandler2;

public class Handler extends DefaultHandler2 {
    private boolean isPublisher, isTitle;
    private String isbn, publisher, pubYear, title, srchText;

    public Handler(String srchText) {
        this.srchText = srchText;
    }

    @Override
    public void characters(char[] ch, int start, int length) {
        if (isTitle) {
            title = new String(ch, start, length).trim();
            isTitle = false;
        } else if (isPublisher) {
            publisher = new String(ch, start, length).trim();
            isPublisher = false;
        }
    }
}
```

```java
@Override
public void endElement(String uri, String localName, String qName)
{
    if (!localName.equals("book"))
        return;
    if (!srchText.equals(publisher))
        return;
    System.out.println("title = " + title + ", isbn = " + isbn);
}

@Override
public void error(SAXParseException saxpe)
{
    System.out.println("error() " + saxpe);
}

@Override
public void fatalError(SAXParseException saxpe)
{
    System.out.println("fatalError() " + saxpe);
}

@Override
public void startElement(String uri, String localName, String qName,
                           Attributes attributes)
{
    if (localName.equals("title"))
    {
        isTitle = true;
        return;
    }
    else if (localName.equals("publisher"))
    {
        isPublisher = true;
        return;
    }
    if (!localName.equals("book"))
        return;
    for (int i = 0; i < attributes.getLength(); i++)
        if (attributes.getLocalName(i).equals("isbn"))
            isbn = attributes.getValue(i);
        else
            if (attributes.getLocalName(i).equals("pubyear"))
                pubYear = attributes.getValue(i);

}

@Override
public void warning(SAXParseException saxpe)
{
    System.out.println("warning() " + saxpe);
}
```
Listing A-65. The DOM Equivalent of SAXSearch and Handler

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class DOMSearch
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java DOMSearch publisher");
            return;
        }

        try
        {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse("books.xml");
            class BookItem
            {
                String title;
                String isbn;
            }
            List<BookItem> bookItems = new ArrayList<BookItem>();
            NodeList books = doc.getElementsByTagName("book");
            for (int i = 0; i < books.getLength(); i++)
            {
                Element book = (Element) books.item(i);
                NodeList children = book.getChildNodes();
                String title = "";
                for (int j = 0; j < children.getLength(); j++)
                {
                    Node child = children.item(j);
                    if (child.getNodeType() == Node.ELEMENT_NODE)
                    {
                        // Process the child element
                    }
                    else if (child.getNodeType() == Node.TEXT_NODE)
                    {
                        // Process the text node
                    }
                }
            }
        }
    }
}
```
if (child.getNodeType() == Node.ELEMENT_NODE) {
    if (child.getNodeName().equals("title"))
        title = child.getFirstChild().getNodeValue().trim();
    else
        if (child.getNodeName().equals("publisher"))
            // Compare publisher name argument (args[0]) with text of
            // publisher's child text node. The trim() method call
            // removes whitespace that would interfere with the
            // comparison.
            if (args[0].equals(child.getFirstChild().getNodeValue().trim()))
                {
                    BookItem bookItem = new BookItem();
                    bookItem.title = title;
                    NamedNodeMap nnm = book.getAttributes();
                    Node isbn = nnm.getNamedItem("isbn");
                    bookItem.isbn = isbn.getNodeValue();
                    bookItems.add(bookItem);
                    break;
                }
}
}
}
for (BookItem bookItem: bookItems)
    System.out.println("title = " + bookItem.title + ", isbn = " + bookItem.isbn);
}
catch (IOException ioe)
{
    System.err.println("IOE: " + ioe);
}
catch (SAXException saxe)
{
    System.err.println("SAXE: " + saxe);
}
catch (FactoryConfigurationError fce)
{
    System.err.println("FCE: " + fce);
}
catch (ParserConfigurationException pce)
{
    System.err.println("PCE: " + pce);
}
}
Listing A-66. A Contacts Document with a Titlecased Name Element

```xml
<?xml version="1.0"?>
<contacts>
  <contact>
    <Name>John Doe</Name>
    <city>Chicago</city>
    <city>Denver</city>
  </contact>
  <contact>
    <name>Jane Doe</name>
    <city>New York</city>
  </contact>
  <contact>
    <name>Sandra Smith</name>
    <city>Denver</city>
    <city>Miami</city>
  </contact>
  <contact>
    <name>Bob Jones</name>
    <city>Chicago</city>
  </contact>
</contacts>
```

Listing A-67. Searching for name or Name Elements via a Multiple Selection

```java
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XPathSearch
{
    public static void main(String[] args)
    {
        try
```
62. Listing A-68 and Listing A-69 present the books.xsl document stylesheet file and MakeHTML application that were called for in Chapter 15.

Listing A-68. A Stylesheet for Converting books.xml Content to HTML

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/books">
        <html>
            <head>
                <title>Books</title>
            </head>
            ...
        </html>
    </xsl:template>
</xsl:stylesheet>
```
Listing A-69. Converting books.xml to HTML via a Stylesheet

```java
import java.io.FileReader;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerFactoryConfigurationError;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class MakeHTML {
    public static void main(String[] args) {
        try {
            ...
```
{
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    DocumentBuilder db = dbf.newDocumentBuilder();
    Document doc = db.parse("books.xml");
    TransformerFactory tf = TransformerFactory.newInstance();
    StreamSource ssStyleSheet;
    ssStyleSheet = new StreamSource(new FileReader("books.xsl");)
    Transformer t = tf.newTransformer(ssStyleSheet);
    t.setOutputProperty(OutputKeys.METHOD, "html");
    t.setOutputProperty(OutputKeys.INDENT, "yes");
    Source source = new DOMSource(doc);
    Result result = new StreamResult(System.out);
    t.transform(source, result);
}

} catch (IOException ioe) {
    System.err.println("IOE: " + ioe);
}

} catch (FactoryConfigurationError fce) {
    System.err.println("FCE: " + fce);
}

} catch (ParserConfigurationException pce) {
    System.err.println("PCE: " + pce);
}

} catch (SAXException saxe) {
    System.err.println("SAXE: " + saxe);
}

} catch (TransformerConfigurationException tce) {
    System.err.println("TCE: " + tce);
}

} catch (TransformerException te) {
    System.err.println("TE: " + te);
}

} catch (TransformerFactoryConfigurationError tfce) {
    System.err.println("TFCE: " + tfce);
}

}
Chapter 16: Focusing on Odds and Ends

1. The two enhancements to numeric literals introduced by Java 7 are binary integer literals and underscores in numeric literals.

2. The diamond operator is an empty pair of angle brackets (<> ) that serves as shorthand for specifying the actual type arguments when instantiating a generic type.

3. The answer is true: when multiple exception types are listed in a catch block’s header, the parameter is implicitly regarded as final.

4. The statement that Java 7 uses to implement automatic resource management is try-with-resources.

5. A classloader is an object that dynamically loads compiled classes and other reference types (classes, for short) from classfiles, Java Archive (JAR) files, URLs, and other sources into memory.

6. A classloader’s purpose is to insulate the virtual machine from filesystems, networks, and so on.

7. When the virtual machine starts running, the available classloaders are bootstrap, extension, and system.

8. The answer is false: the abstract ClassLoader class is the ultimate root class for all classloaders (including extension and system) except for bootstrap.

9. The bootstrap classloader is the root classloader.

10. The current classloader is the classloader that loads the class to which the currently executing method belongs.

11. The context classloader is the classloader associated with the current thread.

12. Central to ClassLoader are its Class<?> loadClass(String name) and protected Class<?> loadClass(String name, boolean resolve) methods, which try to load the class with the specified name.

13. The answer is true: context classloaders can be a source of difficulty. You’ll either observe a thrown ClassNotFoundException instance or you may end up with the wrong version of a class.

14. You can use a classloader to load arbitrary resources (such as images). For example, ClassLoader declares an InputStream getResourceAsStream(String name) method for this purpose.

15. The Console class provides methods to access the character-based console device, if any, associated with the current virtual machine.
16. You obtain the console by calling the System class's `Console console()` class method and then testing the return value for null. If it isn't null, you have obtained the console.

17. The answer is false: `Console`'s `String readLine()` method reads a single line of text (not including line-termination characters) from the console's input stream.

18. A design pattern is a catalogued problem/solution entity that consists of a name, a problem statement, a solution, and a list of consequences.

19. The Strategy pattern lets you define a family of algorithms (such as sorting algorithms), encapsulate each algorithm in its own class, and make these algorithms interchangeable. Each encapsulated algorithm is known as a strategy. At runtime, your application chooses the appropriate algorithm that meets its requirements.

20. Double brace initialization is a special syntax for initializing a newly created object in a compact manner, for example, `new Office() {{addEmployee(new Employee("John Doe"));}};`.

21. The two drawbacks of double brace initialization are a potential bloated number of classfiles and the inability to use Java 7’s diamond operator when instantiating a generic type that’s been subclassed by an anonymous class.

22. A fluent interface is an implementation of an object-oriented API that provides more readable code. It’s normally implemented via method chaining to relay the instruction context of a subsequent method call. The context is defined through the return value of a called method, the context is self-referential in that the new context is equivalent to the last context, and the context is terminated through the return of a void context.

23. Five advantages that immutable classes have over mutable classes are as follows: objects created from immutable classes are thread-safe and there are no synchronization issues; you can share the internals of an immutable class to reduce memory usage and improve performance; immutable classes support failure atomicity, which means that, after throwing an exception, an object is still in a well-defined and usable state, even when the exception occurred during an operation; immutable classes are excellent building blocks for more complex classes; and immutable objects make good `Map` keys and `Set` elements because objects must not change state while in a collection.

24. Four guidelines for making a class immutable are as follows: don’t include setter or other mutator methods in the class design, prevent methods from being overridden, declare all fields `final`, and prevent the class from exposing any mutable state.
25. Internationalization is the process of creating an application that automatically adapts to its current user's culture (without recompilation) so that the user can read text in the user's language and otherwise interact with the application without observing cultural biases.

26. Localization is the adaptation of internationalized software to support a new culture by adding culture-specific elements (such as text strings that have been translated to the culture).

27. A locale is a geographical, political, or cultural region.

28. Java provides the Locale class to represent a locale.

29. An expression for obtaining the Canadian locale is new Locale("en", "CA") or Locale.CANADA.

30. You can change the default locale that's made available to the virtual machine by assigning appropriate values to the user.language and user.country system properties (via the -D command-line option) when you launch the application via the java tool.

31. A resource bundle is a container that holds one or more locale-specific elements, and which are each associated with one and only one locale.

32. The answer is true: each resource bundle family shares a common base name.

33. An application loads its resource bundles by calling the various getBundle() class methods that are located in the abstract ResourceBundle class.

34. MissingResourceException is thrown when a resource bundle cannot be found after an exhaustive search.

35. A property resource bundle is a resource bundle backed by a properties file, which is a text file (with a .properties extension) that stores textual elements as a series of key=value entries. PropertyResourceBundle, a concrete subclass of ResourceBundle, manages property resource bundles.

36. A list resource bundle is a resource bundle backed by a classfile, which describes a concrete subclass of ListResourceBundle (an abstract subclass of ResourceBundle).

37. The answer is false: when a property resource bundle and a list resource bundle have the same complete resource bundle name, the list resource bundle takes precedence over the property resource bundle.

38. ResourceBundle provides the void clearCache() and void clearCache(ClassLoader loader) class methods that make it possible to design a server application that clears out all cached resource bundles upon command.
39. Java 6 reworked ResourceBundle to depend on a nested Control class because ResourceBundle's `getBundle()` methods were previously hardwired to look for resource bundles, and resource bundlers were always cached. This lack of flexibility prevented you from performing tasks such as obtaining resource data from sources other than properties files and classfiles (such as an XML file or a database). The nested Control class provides several callback methods that are invoked during the resource bundle search-and-load process. By overriding specific callback methods, you can achieve the desired flexibility. When none of these methods are overridden, the `getBundle()` methods behave as they always have.

40. The `BreakIterator` class lets you detect text boundaries.

41. The `Collator` class lets you make reliable string comparisons, which is especially important for languages where the relative order of their characters doesn't correspond to the Unicode values of these characters.

42. A date is a recorded temporal moment, a time zone is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a calendar is a system of organizing the passage of time.

43. The `Date` class represents a date as a positive or negative milliseconds value that's relative to the Unix Epoch (January 1, 1970 GMT).

44. You obtain a calendar for the default locale that uses a specific time zone by invoking the `Calendar.getInstance(TimeZone zone)` factory method.

45. The answer is false: `Calendar` declares a `Date getTime()` method that returns a calendar's time representation as a `Date` instance.

46. The `Format` subclass that lets you obtain formatters that format numbers as currencies, integers, numbers with decimal points, and percentages (and also to parse such values) is `NumberFormat`.

47. You would obtain a date formatter to format the time portion of a date in a particular style for the default locale by invoking the `DateFormat getTimeInstance(int style)` factory method.

48. The difference between a simple message and a compound message is as follows: a simple message consists of static (unchanging) text, whereas a compound message consists of static text and variable (changeable) data.

49. To format a simple message, you obtain its text from a resource bundle and then display this text to the user. For a compound message, you obtain a pattern (template) for the message from a property resource bundle, pass this pattern along with the variable data to a message formatter to create a simple message, and display this message's text.

50. Java provides the `MessageFormatter` class to format simple and compound messages.
51. The class that you should use to format a compound message that contains singular and plural words is **ChoiceFormat**.

52. The answer is true: the **Format** class declares **parseObject()** methods for parsing strings into objects.

53. The package associated with Java’s Logging API is **java.util.logging**.

54. The **Logger** class is the entry-point into the Logging API.

55. The standard set of log level constants provided by the **Level** class are **SEVERE**, **WARNING**, **INFO**, **CONFIG**, **FINE**, **FINER**, and **FINEST**.

56. The answer is false: loggers default to outputting log records for severe, warning, and informational log levels only.

57. You obtain a logger by invoking one of **Logger’s get Logger()** or **getAnonymousLogger()** class methods.

58. You obtain a logger’s nearest parent logger by invoking **Logger’s get Logger()** method.

59. The four categories of message-logging methods are **log()**, **logp()**, **logrb()**, and miscellaneous (such as **info()**).

60. The entering() and exiting() methods log messages at the **Level.FINER** log level.

61. You change a logger’s log level by invoking **Logger’s void set Level(Level new Level)** method.

62. The **Filter** interface provides the **boolean is Loggable(LogRecord record)** method for further filtering log records regardless of their log levels.

63. You obtain the name of the method that’s the source of a log record by invoking **LogRecord’s String getSourceMethodName()** method.

64. The logging framework sends a log record to its ultimate destination (such as the console) by invoking a concrete handler’s overriding **void publish(LogRecord record)** method.

65. You obtain a logger’s registered handlers by invoking **Logger’s Handler[] get Handlers()** method.

66. The answer is false: **FileHandler’s default formatter** is an instance of **XMLFormatter**.

67. The **LogManager** class manages a hierarchical namespace of all named **Logger** objects and maintains the configuration properties of the logging framework.

68. The Logging API’s logging methods never throw exceptions; it would be burdensome to force developers to wrap all of their logging calls in try/catch constructs. Besides, how would the application recover from an exception?
in the logging framework? Because Handler classes such as FileHandler and StreamHandler can experience I/O exceptions, the Logging API provides the ErrorManager class so that a handler can report an exception instead of discarding it.

69. The Preferences API lets you store preferences in a hierarchical manner so that you can avoid name collisions. Because this API is backend-neutral, it doesn’t matter where the preferences are stored (a file, a database, or [on Windows platforms] the registry); you don’t have to hardcode file names and locations. Also, there are no text files that can be modified, and Preferences can be used on diskless platforms.

70. The Preferences API manages preferences by using trees of nodes. These nodes are the analogue of a hierarchical filesystem’s directories. Also, preference name/value pairs stored under these nodes are the analogues of a directory’s files. You navigate these trees in a similar manner to navigating a filesystem: specify an absolute path starting from the root node (/) to the node of interest, such as /window/location and /window/size.

71. The difference between the system preference tree and the user preference tree is as follows: all users share the system preference tree, whereas the user preference tree is specific to a single user, which is generally the person who logged into the underlying platform.

72. The package assigned to the Preferences API is java.util.prefs. This package contains three interfaces (NodeChangeListener, PreferencesChangeListener, and PreferencesFactory), four regular classes (AbstractPreferences, NodeChangeEvent, PreferenceChangeEvent, and Preferences), and two exception classes (BackingStoreException and InvalidPreferencesFormatException).

73. The Preferences API’s Preferences class is the entry point.

74. You obtain the root node of the system preference tree by invoking Preferences’s Preferences systemRoot() class method.

75. The Runtime class provides Java applications with access to their runtime environment. You obtain an instance of this class by invoking its Runtime getRuntime() class method.

76. Runtime provides several exec() methods for executing other applications. For example, Process exec(String program) executes the program named program in a separate native process. The new process inherits the environment of the method’s caller, and a Process object is returned to allow communication with the new process. IOException is thrown when an I/O error occurs.
77. The Java Native Interface is a native programming interface that lets Java code running in a virtual machine interoperate with native libraries written in other languages (such as C, C++, or even assembly language). The JNI is a bridge between the virtual machine and the underlying platform to which the native libraries target.

78. A hybrid library is a combination of a Java class and a native interface library, which is a library that provides a native function counterpart for each declared native method, and it can communicate directly with platform-specific libraries.

79. The JNI throws UnsatisfiedLinkError when you try to load a nonexistent library.

80. The package for working with ZIP archives is java.util.zip.

81. Each stored file in a ZIP archive is known as a ZIP entry, which is represented by the ZipEntry class.

82. The answer is true: the name of a stored file in a ZIP archive cannot exceed 65,536 bytes.

83. ZipOutputStream writes ZIP entries (compressed as well as uncompressed) to a ZIP archive.

84. ZipInputStream reads ZIP entries (compressed as well as uncompressed) from a ZIP archive.

85. ZipFile appears to be an alias for ZipInputStream.

86. The package for working with JAR files is java.util.jar.

87. The manifest is a special file named MANIFEST.MF that stores information about the contents of the JAR file. This file is located in the JAR file’s META-INF directory.

88. The answer is true: when creating a manifest for a JAR output stream, you must store MANIFEST_VERSION; otherwise, you’ll observe a thrown exception at runtime.

89. Listing A-70 presents the SpanishCollation application that was called for in Chapter 16.

Listing A-70. Outputting Spanish Words According to This Language’s Current Collation Rules Followed by Its Traditional Collation Rules

```java
import java.text.Collator;
import java.text.ParseException;
import java.text.RuleBasedCollator;
import java.util.Arrays;
import java.util.Locale;
```

www.it-ebooks.info
public class SpanishCollation
{
    public static void main(String[] args)
    {
        String[] words =
        {
            "ñango", // weak
            "llamado", // called
            "lunes", // monday
            "champán", // champagne
            "clamor", // outcry
            "cerca", // near
            "nombre", // name
            "chiste", // joke
        };
        Locale locale = new Locale("es", "");
        Collator c = Collator.getInstance(locale);
        Arrays.sort(words, c);
        for (String word: words)
            System.out.println(word);
        System.out.println();
        // Define the traditional Spanish sort rules.
        String upperNTilde = new String ("\u00D1");
        String lowerNTilde = new String ("\u00F1");
        String spanishRules = 
            "< a, A < b, B < c, C < ch, Ch, CH < d, D < e, E " +
            "< f, F < g, G < h, H < i, I < j, J < k, K < l, L < ll, " +
            "ll, Ll, LL < m, M < n, N < " + lowerNTilde + "," +
            upperNTilde + " < o, O < p, P < q, Q < r, R < s, S < " +
            "t, T < u, U < v, V < w, W < x, X < y, Y < z, Z";
        try
        {
            c = new RuleBasedCollator(spanishRules);
            Arrays.sort(words, c);
            for (String word: words)
                System.out.println(word);
        }
        catch (ParseException pe)
        {
            System.err.println(pe);
        }
    }
}

Compile Listing A-70 as follows:
javac SpanishCollation.java

Run this application as follows:
java SpanishCollation
Listing A-71 presents the ZipList application that was called for in Chapter 16.

**Listing A-71. Archive Contents**

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Date;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

public class ZipList {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("usage: java ZipList zipfile");
            return;
        }
        ZipInputStream zis = null;
        try {
            zis = new ZipInputStream(new FileInputStream(args[0]));
            ZipEntry ze;
```
while ((ze = zis.getNextEntry()) != null)
{
    System.out.println(ze.getName());
    System.out.println(" Compressed Size: " + ze.getCompressedSize());
    System.out.println(" Uncompressed Size: " + ze.getSize());
    if (ze.getTime() != -1)
        System.out.println(" Modification Time: " + new Date(ze.getTime()));
    System.out.println();
    zis.closeEntry();
}
}

} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    if (zis != null)
        try {
            zis.close();
        } catch (IOException ioe) {
            assert false; // shouldn't happen in this context
        }
}
Four of a Kind

Application development isn’t an easy task. If you don’t plan carefully before you develop an application, you’ll probably waste your time and money as you endeavour to create it, and waste your users’ time and money when it doesn’t meet their needs.

Caution It’s extremely important to test your software carefully. You could face a lawsuit if malfunctioning software causes financial harm to its users.

In this appendix, I present one technique for developing applications efficiently. I present this technique in the context of a Java application that lets you play a simple card game called Four of a Kind against the computer.

Understanding Four of a Kind

Before sitting down at the computer and writing code, you need to fully understand the problem domain that you are trying to model via that code. In this case, the problem domain is Four of a Kind, and you want to understand how this card game works.

Two to four players play Four of a Kind with a standard 52-card deck. The object of the game is to be the first player to put down four cards that have the same rank (four aces, for example), which wins the game.

The game begins by shuffling the deck and placing it face down. Each player takes a card from the top of the deck. The player with the highest ranked card (king is highest) deals four cards to each player, starting with the player to the dealer’s left. The dealer then starts its turn.

The player examines her cards to determine which cards are optimal for achieving four of a kind. The player then throws away the least helpful card on a discard pile and picks up another card from the top of the deck. (If each card has a different rank, the player randomly selects a card to throw away.) If the player has four of a kind, the player puts down these cards (face up) and wins the game.
Modeling Four of a Kind in Pseudocode

Now that you understand how *Four of a Kind* works, you can begin to model this game. You will not model the game in Java source code because you would get bogged down in too many details. Instead, you will use pseudocode for this task.

Pseudocode is a compact and informal high-level description of the problem domain. Unlike the previous description of *Four of a Kind*, the pseudocode equivalent is a step-by-step recipe for solving the problem. Check out Listing B-1.

**Listing B-1. Four of a Kind Pseudocode for Two Players (Human and Computer)**

1. Create a deck of cards and shuffle the deck.
2. Create empty discard pile.
3. Have each of the human and computer players take a card from the top of the deck.
4. Designate the player with the highest ranked card as the current player.
5. Return both cards to the bottom of the deck.
6. The current player deals four cards to each of the two players in alternating fashion, with the first card being dealt to the other player.
7. The current player examines its current cards to see which cards are optimal for achieving four of a kind. The current player throws the least helpful card onto the top of the discard pile.
8. The current player picks up the deck's top card. If the current player has four of a kind, it puts down its cards and wins the game.
9. Designate the other player as the current player.
10. If the deck has no more cards, empty the discard pile to the deck and shuffle the deck.
11. Repeat at step 7.

Deriving Listing B-1’s pseudocode from the previous description is the first step in achieving an application that implements *Four of a Kind*. This pseudocode performs various tasks including decision making and repetition.

Despite being a more useful guide to understanding how *Four of a Kind* works, Listing B-1 is too high level for translation to Java. Therefore, you must refine this pseudocode to facilitate the translation process. Listing B-2 presents this refinement.

**Listing B-2. Refined Four of a Kind Pseudocode for Two Players (Human and Computer)**

1. deck = new Deck()
2. deck.shuffle()
3. discardPile = new DiscardPile()
4. hCard = deck.deal()
5. cCard = deck.deal()
6. if hCard.rank() == cCard.rank()
   6.1. deck.putBack(hCard)
   6.2. deck.putBack(cCard)
   6.3. deck.shuffle()
   6.4. Repeat at step 4
7. curPlayer = HUMAN
   7.1. if cCard.rank() > hCard.rank()
      7.1.1. curPlayer = COMPUTER
8. deck.putBack(hCard)
9. deck.putBack(cCard)
10. if curPlayer == HUMAN
   10.1. for i = 0 to 3
      10.1.1. cCards[i] = deck.deal()
      10.1.2. hCards[i] = deck.deal()
   else
      10.2. for i = 0 to 3
      10.2.1. hCards[i] = deck.deal()
      10.2.2. cCards[i] = deck.deal()
11. if curPlayer == HUMAN
   11.01. output(hCards)
   11.02. choice = prompt("Identify card to throw away")
   11.03. discardPile.setTopCard(hCards[choice])
   11.04. hCards[choice] = deck.deal()
   11.05. if isFourOfAKind(hCards)
      11.05.1. output("Human wins!")
      11.05.2. putDown(hCards)
      11.05.3. output("Computer’s cards:")
      11.05.4. putDown(cCards)
      11.05.5. End game
   11.06. curPlayer = COMPUTER
   else
      11.07. choice = leastDesirableCard(cCards)
      11.08. discardPile.setTopCard(cCards[choice])
      11.09. cCards[choice] = deck.deal()
      11.10. if isFourOfAKind(cCards)
         11.10.1. output("Computer wins!")
         11.10.2. putDown(cCards)
         11.10.3. End game
      11.11. curPlayer = HUMAN
12. if deck.isEmpty()
   12.1. if discardPile.topCard() != null
      12.1.1. deck.putBack(discardPile.getTopCard())
      12.1.2. Repeat at step 12.1.
   12.2. deck.shuffle()
13. Repeat at step 11.

In addition to being longer than Listing B-1, Listing B-2 shows the refined pseudocode becoming more like Java. For example, Listing B-2 reveals Java expressions (such as new Deck(), to create a Deck object), operators (such as ==, to compare two values for equality), and method calls (such as deck.isEmpty(), to call deck’s isEmpty() method to return a Boolean value indicating whether [true] or not [false] the deck identified by deck is empty of cards).

**Converting Pseudocode to Java Code**

Now that you’ve had a chance to absorb Listing B-2’s Java-like pseudocode, you’re ready to examine the process of converting that pseudocode to Java source code. This process consists of a couple of steps.

The first step in converting Listing B-2’s pseudocode to Java involves identifying important components of the game’s structure and implementing these components as classes, which I formally introduced in Chapter 3.
Apart from the computer player (which is implemented via game logic), the important components are card, deck, and discard pile. I represent these components via Card, Deck, and DiscardPile classes. Listing B-3 presents Card.

Listing B-3. Merging Suits and Ranks into Cards

```java
/**
 * Simulating a playing card.
 *
 * @author Jeff Friesen
 */

public enum Card {
    ACE_OF_CLUBS(Suit.CLUBS, Rank.ACE),
    TWO_OF_CLUBS(Suit.CLUBS, Rank.TWO),
    THREE_OF_CLUBS(Suit.CLUBS, Rank.THREE),
    FOUR_OF_CLUBS(Suit.CLUBS, Rank.FOUR),
    FIVE_OF_CLUBS(Suit.CLUBS, Rank.FIVE),
    SIX_OF_CLUBS(Suit.CLUBS, Rank.SIX),
    SEVEN_OF_CLUBS(Suit.CLUBS, Rank.SEVEN),
    EIGHT_OF_CLUBS(Suit.CLUBS, Rank.EIGHT),
    NINE_OF_CLUBS(Suit.CLUBS, Rank.NINE),
    TEN_OF_CLUBS(Suit.CLUBS, Rank.TEN),
    JACK_OF_CLUBS(Suit.CLUBS, Rank.JACK),
    QUEEN_OF_CLUBS(Suit.CLUBS, Rank.QUEEN),
    KING_OF_CLUBS(Suit.CLUBS, Rank.KING),
    ACE_OF_DIAMONDS(Suit.DIAMONDS, Rank.ACE),
    TWO_OF_DIAMONDS(Suit.DIAMONDS, Rank.TWO),
    THREE_OF_DIAMONDS(Suit.DIAMONDS, Rank.THREE),
    FOUR_OF_DIAMONDS(Suit.DIAMONDS, Rank.FOUR),
    FIVE_OF_DIAMONDS(Suit.DIAMONDS, Rank.FIVE),
    SIX_OF_DIAMONDS(Suit.DIAMONDS, Rank.SIX),
    SEVEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.SEVEN),
    EIGHT_OF_DIAMONDS(Suit.DIAMONDS, Rank.EIGHT),
    NINE_OF_DIAMONDS(Suit.DIAMONDS, Rank.NINE),
    TEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.TEN),
    JACK_OF_DIAMONDS(Suit.DIAMONDS, Rank.JACK),
    QUEEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.QUEEN),
    KING_OF_DIAMONDS(Suit.DIAMONDS, Rank.KING),
    ACE_OF_HEARTS(Suit.HEARTS, Rank.ACE),
    TWO_OF_HEARTS(Suit.HEARTS, Rank.TWO),
    THREE_OF_HEARTS(Suit.HEARTS, Rank.THREE),
    FOUR_OF_HEARTS(Suit.HEARTS, Rank.FOUR),
    FIVE_OF_HEARTS(Suit.HEARTS, Rank.FIVE),
    SIX_OF_HEARTS(Suit.HEARTS, Rank.SIX),
    SEVEN_OF_HEARTS(Suit.HEARTS, Rank.SEVEN),
    EIGHT_OF_HEARTS(Suit.HEARTS, Rank.EIGHT),
    NINE_OF_HEARTS(Suit.HEARTS, Rank.NINE),
    TEN_OF_HEARTS(Suit.HEARTS, Rank.TEN),
    JACK_OF_HEARTS(Suit.HEARTS, Rank.JACK),
    QUEEN_OF_HEARTS(Suit.HEARTS, Rank.QUEEN),
```
KING_OF_HEARTS(Suit.HEARTS, Rank.KING),
ACE_OF SPADES(Suit.SPADES, Rank.ACE),
TWO_OF SPADES(Suit.SPADES, Rank.TWO),
THREE_OF SPADES(Suit.SPADES, Rank.THREE),
FOUR_OF SPADES(Suit.SPADES, Rank.FOUR),
FIVE_OF SPADES(Suit.SPADES, Rank.FIVE),
SIX_OF SPADES(Suit.SPADES, Rank.SIX),
SEVEN_OF SPADES(Suit.SPADES, Rank.SEVEN),
EIGHT_OF SPADES(Suit.SPADES, Rank.EIGHT),
NINE_OF SPADES(Suit.SPADES, Rank.NINE),
TEN_OF SPADES(Suit.SPADES, Rank.TEN),
JACK_OF SPADES(Suit.SPADES, Rank.JACK),
QUEEN_OF SPADES(Suit.SPADES, Rank.QUEEN),
KING_OF SPADES(Suit.SPADES, Rank.KING);

private Suit suit;
/**
 *  Return <code>Card</code>'s suit.
 *  @return <code>CLUBS</code>, <code>DIAMONDS</code>, <code>HEARTS</code>,
 *  or <code>SPADES</code>
 */

class Card()
{
    suit = suit;
    rank = rank;
}

www.it-ebooks.info
Listing B-3 begins with a Javadoc comment that's used to briefly describe the subsequently declared
Card class and identify this class's author. (I briefly introduced Javadoc comments in Chapter 2.)

Card is an example of an enum, which is a special kind of class that I discussed in Chapter 6. If you
haven't read that chapter, think of Card as a place to create and store Card objects that identify all 52
cards that make up a standard deck.

Card declares a nested Suit enum. (I discussed nesting in Chapter 5.) A card’s suit denotes its
membership. The only legal Suit values are CLUBS, DIAMONDS, HEARTS, and SPADES.

Card also declares a nested Rank enum. A card’s rank denotes its value: ACE, TWO, THREE, FOUR, FIVE,
SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, and KING are the only legal Rank values.

A Card object is created when Suit and Rank objects are passed to its constructor. (I discussed
constructors in Chapter 3.) For example, KING_OF_HEARTS(Suit.HEARTS, Rank.KING) combines
Suit.HEARTS and Rank.KING into KING_OF_HEARTS.

Card provides a rank() method for returning a Card’s Rank object. Similarly, Card provides a suit() method for returning a Card’s Suit object. For example, KING_OF_HEARTS.rank() returns Rank.KING,
and KING_OF_HEARTS.suit() returns Suit.HEARTS.
Listing B-4 presents the Java source code to the Deck class, which implements a deck of 52 cards.

Listing B-4. Pick a Card, Any Card

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Simulate a deck of cards.
 * @author Jeff Friesen
 */

class Deck {
    private Card[] cards = new Card[] {
        Card.ACE_OF_CLUBS,
        Card.TWO_OF_CLUBS,
        Card.TWO_OF_CLUBS,
        Card.FOUR_OF_CLUBS,
        Card.FIVE_OF_CLUBS,
        Card.SIX_OF_CLUBS,
        Card.SEVEN_OF_CLUBS,
        Card.EIGHT_OF_CLUBS,
        Card.NINE_OF_CLUBS,
        Card.TEN_OF_CLUBS,
        Card.JACK_OF_CLUBS,
        Card.QUEEN_OF_CLUBS,
        Card.KING_OF_CLUBS,
        Card.ACE_OF_DIAMONDS,
        Card.TWO_OF_DIAMONDS,
        Card.TWO_OF_DIAMONDS,
        Card.TWO_OF_DIAMONDS,
        Card.FOUR_OF_DIAMONDS,
        Card.FIVE_OF_DIAMONDS,
        Card.SIX_OF_DIAMONDS,
        Card.SEVEN_OF_DIAMONDS,
        Card.EIGHT_OF_DIAMONDS,
        Card.NINE_OF_DIAMONDS,
        Card.TEN_OF_DIAMONDS,
        Card.JACK_OF_DIAMONDS,
        Card.QUEEN_OF_DIAMONDS,
        Card.KING_OF_DIAMONDS,
        Card.ACE_OF_HEARTS,
        Card.TWO_OF_HEARTS,
        Card.TWO_OF_HEARTS,
        Card.TWO_OF_HEARTS,
        Card.FOUR_OF_HEARTS,
        Card.FIVE_OF_HEARTS,
        Card.SIX_OF_HEARTS,
        Card.SEVEN_OF_HEARTS,
        Card.EIGHT_OF_HEARTS,
```
private List<Card> deck;

/**
 * Create a <code>Deck</code> of 52 <code>Card</code> objects. Shuffle
 * these objects.
 */

public Deck()
{
    deck = new ArrayList<Card>();
    for (int i = 0; i < cards.length; i++)
    {
        deck.add(cards[i]);
        cards[i] = null;
    }
    Collections.shuffle(deck);
}

/**
 * Deal the <code>Deck</code>'s top <code>Card</code> object.
 *
 * @return the <code>Card</code> object at the top of the
 * <code>Deck</code>
 */

public Card deal()
{
    return deck.remove(0);
}
/**
 * Return an indicator of whether or not the <code>Deck</code> is empty.
 * @return true if the <code>Deck</code> contains no <code>Card</code> objects; otherwise, false
 */

public boolean isEmpty()
{
    return deck.isEmpty();
}

/**
 * Put back a <code>Card</code> at the bottom of the <code>Deck</code>.
 * @param card <code>Card</code> object being put back
 */

public void putBack(Card card)
{
    deck.add(card);
}

/**
 * Shuffle the <code>Deck</code>.
 */

public void shuffle()
{
    Collections.shuffle(deck);
}

Deck initializes a private cards array to all 52 Card objects. Because it’s easier to implement Deck via a list that stores these objects, Deck’s constructor creates this list and adds each Card object to the list. (I discussed List and ArrayList in Chapter 9.)

Deck also provides deal(), isEmpty(), putBack(), and shuffle() methods to deal a single Card from the Deck (the Card is physically removed from the Deck), determine whether or not the Deck is empty, put a Card back into the Deck, and shuffle the Deck’s Cards.

Listing B-5 presents the source code to the DiscardPile class, which implements a discard pile on which players can throw away a maximum of 52 cards.

Listing B-5. A Garbage Dump for Cards

import java.util.ArrayList;
import java.util.List;

/**
 * Simulate a pile of discarded cards.
 * @author Jeff Friesen
 */
public class DiscardPile {
    private Card[] cards;
    private int top;

    /**
     * Create a <code>DiscardPile</code> that can accommodate a maximum of 52
     * <code>Card</code>s. The <code>DiscardPile</code> is initially empty.
     */
    public DiscardPile() {
        cards = new Card[52]; // Room for entire deck on discard pile (should
        // never happen).
        top = -1;
    }

    /**
     * Return the <code>Card</code> at the top of the <code>DiscardPile</code>.
     * @return <code>Card</code> object at top of <code>DiscardPile</code> or
     * null if <code>DiscardPile</code> is empty
     */
    public Card getTopCard() {
        if (top == -1)
            return null;
        Card card = cards[top];
        cards[top--] = null;
        return card;
    }

    /**
     * Set the <code>DiscardPile</code>\'s top card to the specified
     * <code>Card</code> object.
     * @param card <code>Card</code> object being thrown on top of the
     * <code>DiscardPile</code>
     */
    public void setTopCard(Card card) {
        cards[++top] = card;
    }
}
DiscardPile implements a discard pile on which to throw Card objects. It implements the discard pile via a stack metaphor: the last Card object thrown on the pile sits at the top of the pile and is the first Card object to be removed from the pile.

This class stores its stack of Card objects in a private cards array. I found it convenient to specify 52 as this array’s storage limit because the maximum number of Cards is 52. (Game play will never result in all Cards being stored in the array.)

Along with its constructor, DiscardPile provides getTopCard(), setTopCard(), and topCard() methods to remove and return the stack's top Card, store a new Card object on the stack as its top Card, and return the top Card without removing it from the stack.

The constructor demonstrates a single-line comment, which starts with the // character sequence. This comment documents that the cards array has room to store the entire Deck of Cards. I formally introduced single-line comments in Chapter 2.

The second step in converting Listing B-2’s pseudocode to Java involves introducing a FourOfAKind class whose main() method contains the Java code equivalent of this pseudocode. Listing B-6 presents FourOfAKind.

Listing B-6. FourOfAKind Application Source Code

```java
/**
 * This codeFourOfAKind() implements a card game that is played between two players: one human player and the computer. You play this game with a standard 52-card deck and attempt to beat the computer by being the first player to put down four cards that have the same rank (four aces, for example), and win.
 *
 * <p>
 * The game begins by shuffling the deck and placing it face down. Each player takes a card from the top of the deck. The player with the highest ranked card (king is highest) deals four cards to each player starting with the other player. The dealer then starts its turn.
 */
```
The player examines its cards to determine which cards are optimal for achieving four of a kind. The player then throws away one card on a discard pile and picks up another card from the top of the deck. If the player has four of a kind, the player puts down these cards (face up) and wins the game.

@author Jeff Friesen
@version 1.0
*/

public class FourOfAKind
{
  /**
   * Human player
   */
  final static int HUMAN = 0;

  /**
   * Computer player
   */
  final static int COMPUTER = 1;

  /**
   * Application entry point.
   *
   * @param args array of command-line arguments passed to this method
   */
  public static void main(String[] args)
  {
    System.out.println("Welcome to Four of a Kind!");
    Deck deck = new Deck(); // Deck automatically shuffled
    DiscardPile discardPile = new DiscardPile();
    Card hCard;
    Card cCard;
    while (true)
    {
      hCard = deck.deal();
      cCard = deck.deal();
      if (hCard.rank() != cCard.rank())
        break;
      deck.putBack(hCard);
      deck.putBack(cCard);
      deck.shuffle(); // prevent pathological case where every successive pair of cards have the same rank
    } // pair of cards have the same rank
    int curPlayer = HUMAN;
    if (cCard.rank().ordinal() > hCard.rank().ordinal())
      curPlayer = COMPUTER;
    deck.putBack(hCard);
hCard = null;
decck.putBack(cCard);
cCard = null;
Card[] hCards = new Card[4];
Card[] cCards = new Card[4];
if (curPlayer == HUMAN)
  for (int i = 0; i < 4; i++)
    { 
      cCards[i] = deck.deal(); 
      hCards[i] = deck.deal();
    }
else
  for (int i = 0; i < 4; i++)
    { 
      hCards[i] = deck.deal();
      cCards[i] = deck.deal();
    }
while (true)
{
  if (curPlayer == HUMAN)
  {
    showHeldCards(hCards);
    int choice = 0;
    while (choice < 'A' || choice > 'D')
      {
        choice = prompt("Which card do you want to throw away (A, B, " +
          "C, D)? ");
        switch (choice)
          {
            case 'a': choice = 'A'; break;
            case 'b': choice = 'B'; break;
            case 'c': choice = 'C'; break;
            case 'd': choice = 'D';
          }
  }
  discardPile.setTopCard(hCards[choice - 'A']);
  hCards[choice - 'A'] = deck.deal();
  if (isFourOfAKind(hCards))
    {
      System.out.println();
      System.out.println("Human wins!");
      System.out.println();
      putDown("Human's cards:", hCards);
      System.out.println();
      putDown("Computer's cards:", cCards);
      return; // Exit application by returning from main()
    }
  curPlayer = COMPUTER;
}
else
int choice = leastDesirableCard(cCards);
discardPile.setTopCard(cCards[choice]);
cCards[choice] = deck.deal();
if (isFourOfAKind(cCards))
{
    System.out.println();
    System.out.println("Computer wins!");
    System.out.println();
    putDown("Computer's cards:", cCards);
    return; // Exit application by returning from main()
}
curPlayer = HUMAN;
}
if (deck.isEmpty())
{
    while (discardPile.topCard() != null)
    {
        deck.putBack(discardPile.getTopCard());
        deck.shuffle();
    }
}

/**
 * Determine if the <code>Card</code> objects passed to this method all
 * have the same rank.
 *
 * @param cards array of <code>Card</code> objects passed to this method
 *
 * @return true if all <code>Card</code> objects have the same rank;
 * otherwise, false
 */
static boolean isFourOfAKind(Card[] cards)
{
    for (int i = 1; i < cards.length; i++)
    {
        if (cards[i].rank() != cards[0].rank())
            return false;
    }
    return true;
}

/**
 * Identify one of the <code>Card</code> objects that is passed to this
 * method as the least desirable <code>Card</code> object to hold onto.
 *
 * @param cards array of <code>Card</code> objects passed to this method
 *
 * @return 0-based rank (ace is 0, king is 13) of least desirable card
 */
static int leastDesirableCard(Card[] cards)
{
    int[] rankCounts = new int[13];
    for (int i = 0; i < cards.length; i++)
        rankCounts[cards[i].rank().ordinal()]++;
    int minCount = Integer.MAX_VALUE;
    int minIndex = -1;
    for (int i = 0; i < rankCounts.length; i++)
        if (rankCounts[i] < minCount && rankCounts[i] != 0)
        {
            minCount = rankCounts[i];
            minIndex = i;
        }
    for (int i = 0; i < cards.length; i++)
        if (cards[i].rank().ordinal() == minIndex)
            return i;
    return 0; // Needed to satisfy compiler (should never be executed)
}

/**
 * Prompt the human player to enter a character.
 * @param msg message to be displayed to human player
 * @return integer value of character entered by user.
 */
static int prompt(String msg)
{
    System.out.print(msg);
    try
    {
        int ch = System.in.read();
        // Erase all subsequent characters including terminating \n newline
        // so that they do not affect a subsequent call to prompt().
        while (System.in.read() != '\n')
            return ch;
    }
    catch (java.io.IOException ioe)
    {
    }
    return 0;
}

/**
 * Display a message followed by all cards held by player. This output
 * simulates putting down held cards.
 * @param msg message to be displayed to human player
 * @param cards array of <code>Card</code> objects to be identified
 */
static void putDown(String msg, Card[] cards)
{
    System.out.println(msg);
    for (int i = 0; i < cards.length; i++)
        System.out.println(cards[i]);
}

/**
 * Identify the cards being held via their Card objects on
 * separate lines. Prefix each line with an uppercase letter starting with
 * A.
 * @param cards array of Card objects to be identified
 */
static void showHeldCards(Card[] cards)
{
    System.out.println();
    System.out.println("Held cards:");
    for (int i = 0; i < cards.length; i++)
        if (cards[i] != null)
            System.out.println((char) ('A' + i) + ". " + cards[i]);
    System.out.println();
}

Listing B-6 follows the steps outlined by and expands on Listing B-2’s pseudocode. Because of the various comments, I don’t have much to say about this listing. However, there are a couple of items that deserve mention:

- Card’s nested Rank enum stores a sequence of 13 Rank objects beginning with ACE and ending with KING. These objects cannot be compared directly via > to determine which object has the greater rank. However, their integer-based ordinal (positional) values can be compared by calling the Rank object’s ordinal() method. For example, Card.ACE_OF_SPADES.rank().ordinal() returns 0 because ACE is located at position 0 within Rank’s list of Rank objects, and Card.KING_OF_CLUBS.rank().ordinal() returns 12 because KING is located at the last position in this list.

- The leastDesirableCard() method counts the ranks of the Cards in the array of four Card objects passed to this method and stores these counts in a rankCounts array. For example, given two of clubs, ace of spades, three of clubs, and ace of diamonds in the array passed to this method, rankCounts identifies one two, two aces, and one three. This method then searches rankCounts from smallest index (representing ace) to largest index (representing king) for the first smallest nonzero count (there might be a tie, as in one two and one three)—a zero count represents no Cards having that rank in the array of Card objects. Finally, the method searches the array of Card objects to identify the object whose rank ordinal matches the index of the smallest nonzero count and returns the index of this Card object. This behavior implies that the least desirable card is always the smallest ranked card. For example, given two of spades, three of diamonds, five of spades, and nine of clubs, two of spades is least desirable because it has the smallest rank.
Also, when there are multiple cards of the same rank, and when this rank is smaller than the rank of any other card in the array, this method will choose the first (in a left-to-right manner) of the multiple cards having the same rank as the least desirable card. For example, given (in this order) two of spades, two of hearts, three of diamonds, and jack of hearts, two of spades is least desirable because it's the first card with the smallest rank. However, when the rank of the multiple cards isn't the smallest, another card with the smallest rank is chosen as least desirable.

The JDK provides a javadoc tool that extracts all Javadoc comments from one or more source files and generates a set of HTML files containing this documentation in an easy-to-read format. These files serve as the program's documentation.

For example, suppose that the current directory contains Card.java, Deck.java, DiscardPile.java, and FourOfAKind.java. To generate HTML based on the Javadoc comments that appear in these files, specify the following command:

```
javadoc *.java
```

The javadoc tool responds by outputting the following messages:

```
Loading source file Card.java...
Loading source file Deck.java...
Loading source file DiscardPile.java...
Loading source file FourOfAKind.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_06
Building tree for all the packages and classes...
Generating \Card.html...
Generating \Card.Rank.html...
Generating \Card.Suit.html...
Generating \Deck.html...
Generating \DiscardPile.html...
Generating \FourOfAKind.html...
Generating \package-frame.html...
Generating \package-summary.html...
Generating \package-tree.html...
Generating \constant-values.html...
Building index for all the packages and classes...
Generating \overview-tree.html...
Generating \index-all.html...
Generating \deprecated-list.html...
Building index for all classes...
Generating \allclasses-frame.html...
Generating \allclasses-noframe.html...
Generating \index.html...
Generating \help-doc.html...
```

Furthermore, it generates a series of files, including the index.html entry-point file. If you point your web browser to this file, you should see a page that is similar to the page shown in Figure B-1.
Note that javadoc defaults to generating HTML-based documentation for public classes and public/protected members of classes. You learned about public classes and public/protected members of classes in Chapter 3.

For this reason, FourOfAKind’s documentation reveals only the public main() method. It doesn’t reveal isFourOfAKind() and the other package-private methods. If you want to include these methods in the documentation, you must specify -package with javadoc:

```bash
doc -package *.java
```

**Note** The standard class library's documentation from Oracle was also generated by javadoc and adheres to the same format.
Compiling, Running, and Distributing FourOfAKind

Unlike Chapter 1’s DumpArgs and EchoText applications, which each consist of one source file, FourOfAKind consists of Card.java, Deck.java, DiscardPile.java, and FourOfAKind.java. You can compile all four source files via the following command line:

```
javac FourOfAKind.java
```

The `javac` tool launches the Java compiler, which recursively compiles the source files of the various classes it encounters during compilation. Assuming successful compilation, you should end up with six classfiles in the current directory.

**Tip** You can compile all Java source files in the current directory by specifying `javac *.java`.

After successfully compiling `FourOfAKind.java` and the other three source files, specify the following command line to run this application:

```
java FourOfAKind
```

In response, you see an introductory message and the four cards that you are holding. The following output reveals a single session:

```
Welcome to Four of a Kind!

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FOUR_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FOUR_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES
```

www.it-ebooks.info
Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FIVE_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. JACK_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES
Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF SPADES
C. SIX_OF_HEARTS
D. SIX_OF SPADES

Which card do you want to throw away (A, B, C, D)? B

Human wins!

Human's cards:
SIX_OF_CLUBS
SIX_OF_DIAMONDS
SIX_OF_HEARTS
SIX_OF SPADES

Computer's cards:
SEVEN_OF_HEARTS
TEN_OF_HEARTS
SEVEN_OF_CLUBS
SEVEN_OF_DIAMONDS

Although Four of a Kind isn’t much of a card game, you might decide to share the FourOfAKind application with a friend. However, if you forget to include even one of the application’s five supporting classfiles, your friend will not be able to run the application.

You can overcome this problem by bundling FourOfAKind’s six classfiles into a single JAR (Java ARchive) file, which is a ZIP file that contains a special directory and the .jar file extension. You can then distribute this single JAR file to your friend.

The JDK provides the jar tool for working with JAR files. To bundle all six classfiles into a JAR file named FourOfAKind.jar, you could specify the following command line, where c tells jar to create a JAR file and f identifies the JAR file’s name:

```
jar cf FourOfAKind.jar *.class
```

After creating the JAR file, try to run the application via the following command line:

```
java -jar FourOfAKind.jar
```

Instead of the application running, you’ll receive an error message having to do with the java application launcher tool not knowing which of the JAR file’s six classfiles is the main classfile (the file whose class’s main() method executes first).

You can provide this knowledge via a text file that’s merged into the JAR file’s manifest, a special file named MANIFEST.MF that stores information about the contents of a JAR file and which is stored in the JAR file’s META-INF directory. Consider Listing B-7.
Listing B-7. Identifying the Application’s Main Class

Main-Class: FourOfAKind

Listing B-7 tells java which of the JAR’s classfiles is the main classfile. (You must leave a blank line after Main-Class: FourOfAKind.)

The following command line, which creates FourOfAKind.jar, includes m and the name of the text file providing manifest content:

    jar cfm FourOfAKind.jar manifest *.class

This time, java -jar FourOfAKind.jar succeeds and the application runs because java is able to identify FourOfAKind as the main classfile.

Note  Now that you’ve finished this book, you’re ready to dig deeper into Android app development. Check out Beginning Android 4 by Grant Allen (Apress) and Android Recipes Second Edition by Dave Smith and Jeff Friesen (Apress) for guidance. After you’ve learned some more app development basics, perhaps you might consider transforming Four of a Kind into an Android app.
Application program interfaces (APIs) (cont.)
  binary search, 476
  collections class, 477
  empty and nonempty lists, 478
  linear search, 476
  methods, 475
internationalization, 914
Java Android, 9
JDK 7 documentation, 9
legacy collection
  BitSet, 480
dictionary, 480
  hashtable, 480
stack, 480
  vector, 479
logging API, 959
software library, 1
Arraycopy() method, 323
ArrayDeque
  constructors, 447
definition, 447
stack, 447
ArrayList
  constructors, 415
demonstration, 416
description, 415
Assertions
  avoidance, 243
control-flow invariants, 239
description, 236
design-by-contract
class invariants, 243
  postconditions, 242
  preconditions, 240
  enabling and disabling, 244
  internal invariants, 237
  statement forms, 236
  using, 237
Atomic variables
  classes, 533
  getNextID() class method, 534
  synchronization, 534
Autoboxing, 410
Automatic Resource Management, 888

B

Babel Fish, Yahoo!, 925
Basic APIs
  absolute values, byte and short integers, 293
  BigDecimal (see BigDecimal)
  BigInteger, 301
  exploring number, 295
d graphing sine and cosine waves, 290
  math methods, 287
  primitive type wrapper classes
    Boolean, 305
    character, 307
    float and double, 308
    integer, long, short, and byte, 312
  StrictMath and strictfp, 294
String
  constructors and methods, 315–316
description, 314
  iteration, 318
StringBuffer
  constructors and methods, 319
demonstration, 320
StringBuilder, 318, 320
synchronization
  checking account, 336
deadlock, 349
  mutual exclusion, monitors, and locks, 339
  object, 339
  thread-local variables, 352
  utility class, returning unique IDs, 339
  visibility, 340
  waiting and notification, 343
system methods, 322
threads
  description, 324
description, 333
  joining default thread with background
d  thread, 331
  main() method, 327
  methods, 326
  preemptive scheduling, 329
  scheduler, 329
  thread sleep, 330
time slices, 329
uncaught exception handlers, 334
Windows 7 platform, 328
BigDecimal
   based invoice calculations, 300
   constants, 297
   constructors and methods, 298
   floating-point-based invoice
calculations, 296
   RoundingMode constants, 299
   salesTaxPercent, 300
BigInteger
   comparing factorial() methods, 303
   constructors and methods, 302
   description, 301
Bin directory, 766
Boolean primitive type wrapper class, 305
BreakIterators, 935
BufferedOutputStream and
   BufferedInputStream, 588
ByteArrayOutputStream and
   ByteArrayInputStream, 573
ByteBuffer
   allocation and wrapping, 671
   bytes.length capacity, 671
   class methods, 670
   external arrays, 672
   internal byte array, 671
   nondirect byte buffers, 671
   view buffers, 673
Bytes transferring, channels
   demonstration, 705
   methods, 704
   standard output stream, 706
Cast operator
   16-bit unsigned character values, 60
   compound expressions application, 59–60
   description, 59
   primitive-type conversions, 59
Channels, NIO
   asynchronously closeable, 683
   bytes copying, standard input and output
      streams, 684
   data transfer, 681
   definition, 681
   direct byte buffer, 685
   implementation methods, 682
      in depth
         file channels, 689–690, 692
         hole, 692
         locking files (see Locking files)
         read() and write() methods, 691
         scatter/gather, 686–687
         transferring bytes (see Bytes transferring,
            channels)
         unicode values, 689
      interruptible, 683
   I/O classes, 684
   java.nio.channels package, 684
   pipes
      bytes, producing and consuming, 720
      definition, 720
      methods, 720
      receiver task’s run() method, 722
   ReadableByteChannel, 682
   shut down, 683
   socket channels (see Socket channels)
   WritableByteChannel, 682
Character, primitive type wrapper
classes, 307
Charsets. See New I/O (NIO)
Checked exception, 221
Classes
   application, 91
   arrays
      creation syntax, 135
      elements, 134
      initialization, 135
      multidimensional, 136
      object references, 136
      ragged array, 137
Classes (cont.)
behaviors representation via methods, 107
class initializers, 124
class methods
dumpMatrix(), 108
header, 107
method-invocation stack, 109
public static void main(String[] args), 107–108
syntax, 107
declaration, 89
garbage collector, 131
hiding information
contract, 119
helper methods, 119
interface separation, 120
Java support, 120
package-private, 120
private, 120
protected, 120
public, 120
related language feature, 122
revising implementation, 121
initialization order, 128
instance initializers, 126
objects
application, 97
construction syntax, 92
default constructor, 93
explicit constructors, 93
return statement
chaining instance method
calls, 113
instance method, 111
Java, 113
syntax, 111
state and behaviors encapsulation
field-access rules, 106
fields, declaration and access, 99
instance field, declaration and access
(see Instance field)
instance methods, 109
method-invocation rules, 118
overloading methods, 116
pass-by-value passes, 114
recursion, 115
state via fields, 99

Classes, list of
ClassLoader class, 918
EventQueue class, 924
Image class, 923
ImageIcon class, 924
JOptionPane class, 924
ListResourceBundle class, 918, 922
Properties class, 920
PropertyResourceBundle class, 918, 920
Toolkit class, 923
Classic I/O APIs
filesystem (see Filesystem)
java.io package, 620
RandomAccessFile (see RandomAccessFile)
standard I/O, 607
streaming sequences, 620
streams (see Streams)
writers and readers
and InputStream, 610
and OutputStream, 610
byte streams, 607
character sets and encodings, 608
classes, 610
FileWriter and FileReader, 617
Java’s stream classes, 607
OutputStreamWriter and
InputStreamReader, 612
stream characters, 607
Class invariants, 243
ClassLoader.getResource(), 918
Classloaders
and resources, 899
description, 891
mechanics, 892
problems, 897
types, 891
Cloning
deep copying/cloning, 150
shallow cloning, 149
Collators, 939
Collections framework
addAll method, 407
add method, 407
comparable vs. comparator, 402
core interfaces
description, 401
hierarchy, 402
definition, 401
elements, 404
implementation classes, 401–402
iterable and collection, 404
methods, 405
utility classes, 401
Comparator, 404
Compare-and-Swap (CAS), 535
CompareTo() method, 403
Composition, 157
Concurrency utilities
atomic variables (see Atomic Variables)
Compare-and-Swap (CAS), 535–536
concurrent collections
BlockingQueue and ArrayBlockingQueue, 516–517
ConcurrentHashMap, 518
java.util.concurrent package, 515
properties, 515
thread-safe collections, 515
contended synchronization, 535
definition, 488
delayed lock, 535
executors (see Executors)
lock (see Locking framework)
low-level threads API, problems, 487
mutual exclusion, 535
packages, 488
synchronizers (see Synchronizers)
visibility, 535
Console-based application, 902
Console methods, 903
Constant interfaces, 215
    Constants, locale class, 915
    Constructors, locale class, 915
Control-flow invariant, 239
Cookies
    CookieHandler class, 658
    CookieManager
        methods, 658
        objects, 658
description, 657
    HTTP protocol handler, 658
    HTTP response, 657
listing, 659
Covariant return type
definition, 172
demonstration, 172
upcasting and downcasting, 173
createImage(), toolkit class, 923

D
Dalvik Executable (DEX), 27
Databases, accessing
    Java DB (see Java DB)
    Java supports database, 763
    JDBC (see JDBC)
    SQLite, 772
Data Definition Language (DDL), 769
Datagram channels
    client, 716
    methods, 714
    read() and write() methods, 715
    send() method, 716
    server, 716
    stock ticker server, implementation, 717–718
DatagramSocket
    client context, 635
    DatagramPacket(byte[] buf, int length), 633
    DatagramSocket(), 633
    DatagramSocket(int port), 633
definition, 632
    EchoClient, 633
    main() method, 634
    void receive(DatagramPacket dgp), 633
    void send(DatagramPacket dgp), 633
DataOutputStream and DataInputStream, 590
Date constructors and methods, 942
Date formatters, 951
Dates, Time Zones, and Calendars, 941
DDL. See Data Definition Language (DDL)
Deadlock, 349
Deques
    array, 447
definition, 443
    Java documentation, 446
    methods, 443
    queue/equivalent methods, 446
Design patterns, 905
DEX. See Dalvik Executable (DEX)
Diamond operator, 887
Direct byte buffers, 680
Document object model (DOM)
    advantages, SAX, 843
    creating XML documents
Document object model (DOM) (cont.)
  DocumentBuilder, 846–847, 853
  DOMDemo, 853
  outputs, 855
  exploring DOM API, 845–846
  levels, 843
  nodes
    attribute node, 844
    CDATA section node, 844
    child (leaf) node, 843
    comment node, 844
    document fragment node, 844
    document node, 844
    document type node, 844
    element node, 844
    entity node, 844
    entity reference node, 845
    notation node, 845
    parent node, 843
    processing instruction node, 845
    text node, 845
  parsing XML documents
    DOMDemo, 850
    elements, 849
    node methods, 847–848
    XML content, 852
  validation API, 846–847
Document type definition (DTD)
  attribute declaration, 816
  attribute types, 816
  definition, 815
  document type declaration, 817
  element declarations, 815
  general entities, 818
  internal, 817
  parameter entities, 819
  XML parser, 816
Double brace initialization, 909
Downcasting
  array, 171
  description, 169
  DowncastDemo, 170
  trouble, 169
Drawable interfaces
  declaration, 175
  draw() method, 175
  implementation, 176

DumpArgs
  command-line interface, 12
  for loop, 12
  javac, 12
  main(), 11
  source code, 11
  System.out.println(args[i]), 12

E

Early binding, 166
EchoClient, 630–631
EchoServer, 632
EchoText
  Boolean, 13
  command-line arguments, 13–14
  javac, 14
  main(), 13
  outputting text, 13
  standard input stream, 13
  System.in.read(), 14
  System.out.println(), 14
  while loop, 14
EE. See Enterprise Edition (EE)
Embedded ij script session, 770
Employees database creation, 770
Enterprise Edition (EE), 6
Enumerated type
  class, 280
  compile-time type safety, 275
  description, 273
  enhancement, 276
  extending class, 281
  switch statement, 275
  traditional, 273
EnumMap, 467
EnumSet, 425
Equals() method
  hashCode() method, 154
  point objects, 153
  relation, nonnull object references, 152
ErrorManager, 977
Event-dispatching thread, 924
EventQueue.invokeLater(), 924
Exceptions
  cleanup, 229
  custom exception classes, 222
Generics (cont.)
  type safety, 255
  getBundle() class methods property resource bundles, 921
  ResourceBundle class, 917–919
getCookieStore(), 659
default() class method, 658, 915
getMTU() method, 655
getObject() method, 919
Greedy quantifier, 740

H

Handlers and formatters, 971
Hash codes
  hashing, 156
  Java documentation, 156
Hash map
  collision, 454
  constructors, 455
  demonstration, 455
  hash code, 453
  hash function, 453–455
  Image Caches, 459–463
  load factor, 455
  overriding hashCode, 456
HashSet
  constructors, 421
  demonstration, 421
  description, 420
  not overriding hashCode, 422
  overriding hashCode, 423

Implemention inheritance
  appointment calendar class, 158
  vs. composition, 158
  extending appointment calendar class, 159
  forwarding, 160
  fragile base class problem, 160
  Logging Appointment Calendar Class, 161
  wrapper class, 161
InetAddress
  InetAddress[] getAllByName(String host), 624
  InetAddress getAddress(byte[] addr), 624
  InetAddress getByName(String hostName, byte[] ipAddress), 624
  InetAddress getByName(String host), 625
  InetAddress getLocalHost(), 625
InetSocketAddress, 625

Inheritance
  definition, 141
  implementation, 141
  interface, 142
  single and multiple, 141
Initialization, double brace, 909

Instance fields
  Car Class declaration, 102
  constant, 105
  direct access, 102
  Employee class, constant declaration, 105
  initialization via constructors, 103
  nonzero default value, 103
  syntax, 101

Integrated development environment (IDE)
  command-line arguments, 19
  console tab, 18
  DumpArgs, 17
  Eclipse, 15
  Eclipse user interface, 17
  JDK’s tools, 15
  JRE System Library items, 18
  Program arguments, 19
  splash screen, 15
  welcome tab, 16
  workbench, 17
  workspace, 15–16

Interfaces
  agile software development, 182
  declaration, 174
  drawable interface, 182
  extending, 180
fluent interface, 910–911
formalizing class, 174
implementation, 176
inheritance, 177, 180
profiling, 183
Internationalization
APIs
Locale class, 914–915, 1012
ResourceBundle pattern, 917
description, 914, 1011
International Standards Organization (ISO), 915
locales, 915, 1012
resource bundles, 916, 1012
Unicode, 914
International Standards Organization (ISO), 915, 916
Internet Protocol Version 4 (IPv4) address, 622
Internet Protocol Version 6 (IPv6) address, 622
Iterator interface
autoboxing and unboxing, 409
enhanced for loop statement, 408
methods, 408

J, K

JAR API, 1003–1006
Java
APIs, 1
bytecode, 4
C and C++, 2
classfile, 4–5
description, 2
EE, 6
execution environment, 4
IDE (see Integrated development environment (IDE))
implementation sizes, 3
interpretation, 5
JDK (see Java SE Development Kit (JDK))
JIT compilation, 5
JNI, 5
JVM, 4–5
loops and expressions, 3
ME, 6
operators, 2
portability, 6
SE, 6
secure environment, 6
single-and multi-line, 2
source code, 2
standard class library, 4
sun microsystems, 2
Java 6
 boolean setExecutable(boolean executable), 553
 boolean setExecutable(boolean executable, boolean ownerOnly), 553
 boolean setReadable(boolean readable), 553
 boolean setReadable(boolean readable, boolean ownerOnly), 553
 boolean setWritable(boolean writable), 553
 boolean setWritable(boolean writable, boolean ownerOnly), 553
 long getFreeSpace() returns, 546
 long getTotalSpace() returns, 546
 long getUsableSpace() returns, 546
 java.awt.Image class, 923
 java.awt.Toolkit class, 923
 Java Class, 991
 Java code
 computer player, 1130
 merging suits and ranks, 1132
 Java DB
 client/server environment, 768
demos, 767
embedded database engine, start up/shut down, 764
embedded driver, 764
installation and configuration, 766
log directory creation, 765
multiple clients communication, 765
SimpleApp derbyClient command, 768
daJava.lang package, ClassLoader class, 918
Java native interface (JNI), 5, 990
Java SE Development Kit (JDK)
Android (see Android)
APIs, 9
App.class, 8
bracket characters, 9
command-line interface, 8
development tools, 7
documentation, 9
DumpArgs, 11, 13
EchoText, 13–15
HelloWorld.class, 11
Java SE Development Kit (JDK) (cont.)
home directory, 7–8
IDE, 18
Javac, 11
JRE, 7
line feed, 10
main(), 10
public class, 9
source code, 9
standard class library, 8
static and void, 10
String[], args, 10
System.out.println, 10
java.util.jar package, 569
java.util package
ListResourceBundle class, 918
Locale class, 915, 1012
Properties class, 920
PropertyResourceBundle class, 918
ResourceBundle class, 917
Java virtual machine (JVM), 4–5, 519
javax.swing.ImageIcon, 924
JDBC
data sources, 774, 775
DriverManager, 775
driver types, 774
exceptions, 777
Java 7, 775
statements
CallableStatement, 788
definition, 781
executeQuery() method, 784
JDBCDemo application, 781
Metadata (see Metadata)
methods, 781
PreparedStatement, 785
SQL Type/Java type
mappings, 784
URLErrors, 776
URL syntax, 776
Xerial SQLite driver, 776
JIT. See Just-in-time (JIT)
JNI. See Java native interface (JNI)
JOptionPane, 924
Just-in-time (JIT), 5
JVM. See Java virtual machine (JVM)

L
Last-In, First-Out (LIFO) queue, 438
Late binding, description, 166
Learning language fundamentals
application structure, 31–33
comments
description, 33
Javadoc, 34–35, 37
multiline, 33
single-line, 33
description, 31
expressions (see Expressions)
identifiers, 37–38
operators (see Operators)
precedence and associativity
32-bit integer, 69
compound expression, 67
interprets, 69
open and close parentheses, 68
public class CompoundExpressions, 68
statements (see Learning statements)
types
array, 41
data items, 38
integers, 38
user-defined, 41
types (see Primitive types)
variables, 42
Learning statements
assignment, 71
break and labeled break
   employee search application, 82
   infinite loop, 81
   system.out.println, 83
compound, 70
continue statement, 83–84
decision, 71
do-while statement, 80
empty, 81
for statement, 77–78
If-else (see If-else statement)
if statement, 71–72
labeled continue, 84
loop, 76
simple, 70
switch, 75–76
while statement, 78, 80
Lib directory, 766
Linear congruential generator, 360
LineNumberInputStream, 569
LinkedList
    constructors, 418
    linked nodes, demonstration, 418
    node, 417
List
    ArrayList, 415
description, 411
    LinkedList, 416
    ListIterator, methods, 414
    methods, 412
ListResourceBundle class, 918, 922
Locale class
    constants, 915
    constructors, 915
    getOrDefault method, 915
    getISOCountries method, 916
    getISOLanguages method, 916
    Internationalization APIs, 1012
    resource bundles, 917
    setDefault method, 915
Lock, 340
Locking files
    command line, 699
counter variable’s current value, 698
database management system, 693
demonstration, 696
    exclusive and shared, 693
    fileLock’s methods, 693, 695
    main() method, 698
    methods for obtaining, 694
    output, 700
    pattern, 696
    query() method, 699
Locking framework
    condition
        lockstep synchronization, 529
        methods, 524
        synchronization, 524
    lock
        acquisition and release, 520
        methods, 519
        producer and consumer constructors, 528
        shared’s constructor, 528
ReadWriteLock, 529
ReentrantLock
    blocking queue, 523
    constructors, 521
    synchronization, 521
ReentrantReadWriteLock
    constructors, 529
    java Dictionary, 533
    methods, 530
    thread’s working memory, 519
    visibility, 519
Logger hierarchy, 960–962
Logging, 614, 958
LogManager and configuration, 974

M
Mapping files, memory
    demonstration, 702
    map() method, 701
    mode parameter, 701
    output, 704
    position and size parameters, 701
    specified mapping mode, 701
    virtual memory, 700
Maps
    collection views, 450
download entry methods, 451
    EnumMap, 467
    hash map, 453
    IdentityHashMap, 463
    methods, 448
    navigable (see Navigable maps)
    TreeMap, 452
    WeakHashMap, 465
Math’s random() method, 359
ME. See Micro Edition (ME)
Message formatters, 953
Metadata
    catalog and schema, 796
    DatabaseMetaData getMetaData() method, 794
    employee Data Source, 794
    employees existence, determination, 797
    isExist() function, 797
    SQL, 796
    SYS schema stores, 797
Micro Edition (ME), 6
MissingResourceException
getBundle method throwing, 918
getObject method throwing, 919
getString method throwing, 921
Multicasting
datagram packets, 636
definition, 636
group address, 636
int getLength() method, 638
main() method, 638
receiving datagram packets, 637
Multicast, 887
Multipurpose Internet Mail Extensions [MIME] type, 643
Mutual exclusion, 339, 535

N
Native interface library, 992
Navigable maps
definition, 471
methods, 472
tree map, 474
Navigable sets
description, 435
methods, 435
tree set, 437
NetworkInterface
description, 652
enumeration, 654
methods, 652
NetworkInterfaceAddress
enumeration, 656
methods, 655
NetInfo, 656
Network Interface Card (NIC), 623
New I/O (NIO)
buffer
byte ordering, 679
byte-oriented buffer, 666, 669
creation (see ByteBuffer)
definition, 666
direct byte buffers, 680
flipping, 675
in depth, 670
java.nio package, 668
marking, 677–678
methods, 667
properties, 666
subclass operations, 679
writing and reading, 674–675
channels (see Channels, NIO)
charsets
byte order mark, 744
byte sequences, 745
character, 743
codec, 744
coded character set, 743
definition, 744
Internet Assigned Names Authority (IANA), 744
methods, 750
standards, 744
String class, 750
Unicode, 743
definition, 665
formatter
constructors, 752
conversions, 753
demonstration, 754
output, 755
printf(), 756
problem solving ways, 755
syntaxes, 752
void close() method, 753
regular expressions (see Regular expressions)
scanner
constructors, 756
delimiter pattern, 756
output, 758
scanning input, menu context, 757
selectors
attachment, 725
code fragment, 727
definition, 724
interest set, 724
key, 724
nonblocking mode, 723, 725
readiness selection, 724
receiving time, server, 731
selectable channels, 724, 728
selected keys, 726
server socket channel, 730
serving time to clients, 728
source code, 732
streams, 723
NIC. See Network Interface Card (NIC)
Node, 417–419
Nonstatic member class
declaration, 194
description, 193
instance method, 194
ToDo instances, ToDoArray
instance, 195–196
To-Do items, name-description pairs, 195
ToDo List of ToDo instances, 197
Number formatters, 948
Numeric literal enhancements, 886

O
Object-based language, 141
Objects creation. See Classes
Operators
additive, 54–55
array index, 56–57
assignment, 57
bitwise, 57–58
cast (see Cast operator)
conditional, 61–62
equality, 62
logical, 63–64
member access, 64
method call, 65
multiplicative, 65–66
object creation, 66
relational, 66
shift, 66–67
unary minus/plus, 67

P
Packages
and JAR files, 215
description, 206
names, 206
playing, 210
searching
compile-time search, 209
runtime search, 209
statement, 207
Parallelism, 488
Parsing, 956
PartsDB class
append() method, 564
declares constants, 564
description, 564
numRecs() method, 564
RandomAccessFile’s close() method, 564
select() method, 564
update() method, 564
Phantomreference
description, 366
large object’s finalization, 366
main() method, 367
T r, ReferenceQueue<? super T> rq, 366
uses, 366
PipedOutputStream and PipedInputStream, 579
Polymorphism
coercion, 162
description, 162
overloading, 162
parametric, 162
subtype, 163–164
Possessive quantifier, 740–741
PreparedStatement, 785
Primitive types
binary vs. decimal, 39
definition, 38
minimum and maximum values, 40
two byte-integer values, 40
Primitive type wrapper classes
Boolean, 305
definition, 305
PrintStream
description, 568–569
features, 605
line separator, 605
stream classes, 604
System.out.println(), 605
various print() and println() methods, 604
void println() method, 605
PriorityQueue
comparator, 442
description, 438, 440
constructors, 440
integers, 441
ordering elements, 440
Properties files, 919–920
PropertyResourceBundle class, 918–921
Protocol stack, 623
Pseudocode

description, 1128
Java code (see Java code)
refined, 1129

Pseudorandom numbers, 360

Queues

First-In, First-Out (FIFO) queue, 438
Last-In, First-Out (LIFO) queue, 438
methods, 439
PriorityQueue (see PriorityQueue)

Random numbers

array of integers, shuffling, 361
boolean nextBoolean(), 360
double nextDouble() method, 360
double nextGaussian(), 361
float nextFloat(), 361
int nextInt(), 361
int nextInt(int n), 361
linear congruential generator, 360
long nextLong(), 361
pseudorandom numbers, 360
Random(), 360
Random(long seed), 360
random number generators, 359
void nextBytes(byte[] bytes), 361

RandomAccessFile
description, 556
FileDescriptor, 560
file pointer, 557
flat file database, 564
methods, 559
platform-dependent structure, 559
"r", "rw", "rws" and "rwd" modes, 557

References

boolean enqueue(), 364
boolean isEnqueued(), 364
definition, 363
finalize() method, 362
phantomreference (see Phantomreference)
reachability, levels
phantom reachable, 363
softly reachable, 363
strongly reachable, 363
unreachable, 363
weakly reachable, 363
reachable or referenced objects, 362
referencequeue (see Referencequeue)
softreference, 365
subclass, 364
T get(), 364
unreachable (not referenced), 362
void clear(), 364
weakreference, 365

Referencequeue
definition, 363
Reference<? extends T> poll(), 364
Reference<? extends T> remove(), 364
Reference<? extends T> remove(long timeout), 365
ReferenceQueue(), 364

Reflection

accessible objects
boolean isAccessible(), 383
boolean isAnnotationPresent(Class<? extends Annotation> annotationType), 383
<T extends Annotation> T getAnnotation(Class<T> annotationType), 382
void setAccessible(boolean flag), 383
array, 388
class methods, 369
class object
decompileClass() method, 375
decompiler tool, 374
forName() method, 372
getClass() method, 377
constructor
Annotation[] getDeclaredAnnotations(), 378
Class[] getExceptionTypes(), 378
Class[] getParameterTypes(), 378
Class<T> getDeclaringClass(), 378
String getName(), 378
dynamically loaded class
newInstance() method, 377
VideoPlayer class, 378
field
boolean getBoolean(Object object), 379
byte getByte(Object object), 379
char getChar(Object object), 379
class, 380
double getDouble(Object object), 379
float getFloat(Object object), 379
int getInt(Object object), 379
long getLong(Object object), 379
main() method, 380
Object get(Object object), 379
short getShort(Object object), 379

boolean isVarArgs(), 381
class, 381
Class<? getReturnType(), 381
int getModifiers(), 381
main(), 382
Object invoke(Object receiver,
  Object... args), 381

package
  manifest.mf, 386
  methods, 383
  obtaining information, 384

Regular expressions
  boundary matcher, 739
  capturing groups, 738
  character classes, 737
  definition, 732
  Matcher methods, 734
  ox regex, 736
  pattern methods, 733
  PatternSyntaxExceptionMethods, 734
  period metacharacter, 736
  practical, 743
  quantifiers, 740

Reification, 271

Reluctant quantifier, 740–741

Resource-access methods
  getStringArray method, 919
  getString method, 919

ResourceBundle class
  accessing resources, 919
  candidate bundle names, 918–919
  factory methods, 917
  getBundle methods, 917–919
  ListResourceBundle class, 918, 922
  patterns, 917
  properties files, 919
  PropertyResourceBundle class, 918,
    919, 921
  setParent method, 919

Runtime and process, 986

Runtime exception, 221

SAX. See Simple API for XML (SAX)
Scalable vector graphics (SVG), 809

SE. See Standard Edition (SE)

ServerSocket
  accept() method, 628–629
  echoing data, 629
  int port, 628
  int port, int backlog, 628
  int port, int backlog, InetAddress
    localAddress, 628
  ServerSocket(), 628
  setDefault() sets, 915
  setParent() method, 919

Sets
  EnumSet, 425
  HashSet, 420
  navigable (see Navigable sets)
  sorted (see Sorted sets)
  TreeSet (see TreeSet)

Shuffle() methods, 361

Simple API for XML (SAX)
  Handler, 833
  methods
    ContentHandler, 828–829
    DTDHandler, 830
    EntityResolver, 830
    ErrorHandler, 830
    LexicalHandler, 831
    XMLReader object, 825, 827
  SAXDemo, 831–832
  validation, 827

Simple expressions
  application, 48
  array initializers, 45
  boolean and character literal, 44
  counter variable, 46
  escape sequences, 44
  floating-point literal, 44
  integer literal, 44
  literals, 43
  null literal, 44
  one and two-dimensional array, 46
  primitive-type conversions via widening
    conversion rules, 47
  referenced array, 46–47
Simple expressions (cont.)
  string literal, 44
  void method, 43
Simple Mail Transfer Protocol (SMTP), 622
SMTP. See Simple Mail Transfer Protocol (SMTP)
Socket
  echoing data, 629
  InetAddress dstAddress, int dstPort, 626
  InetAddress dstAddress, int dstPort,
      InetAddress localAddr, int localPort, 627
  InputStream getInputStream() methods, 627
  options, 625
  OutputStream getOutputStream() methods, 627
  Socket(), 627
  Socket(Proxy proxy), 627
  String dstName, int dstPort, 627
  void close() method, 627
SocketAddress, 625
Socket channels
  AsynchronousCloseException, 712
  datagram (see Datagram channels)
      definition, 706
      demonstration, 713
      exploring server, 709
  noargument open() method, 712
  nonblocking mode, 707
  TCP/IP stream protocol, 712
  uses, 711
SoftReference
  SoftReference(T r), 365
      T r, ReferenceQueue<? super T> rq, 365
Sorted maps
  definition, 468
  methods, 469
  tree map, 470
Sorted sets
  implementation, 434
  implementing comparable, 433
  methods, 428
  not implementing comparable, 432
  TreeSet, 428, 430
SQLException, 777
SQLite, 772
Standard Edition (SE), 6
Static member class
  class and instance methods, 190
  declaration, 190
description, 189
  multiple implementations, 191
  rectangle implementations, 192
StAX. See Streaming API for XML (StAX)
Strategy implementation, 906
Strategy pattern, 906
Streaming API for XML (StAX), 855
Streams
  BufferedOutputStream and
      BufferedInputStream, 588
  ByteArrayOutputStream and
      ByteArrayInputStream, 573
  DataOutputStream and DataInputStream, 590
  FileOutputStream and FileInputStream, 576
  FilterOutputStream and FilterInputStream, 587
  InputStream methods, 572
  java.io package, 568
  java.util.zip, 569
  LineNumberInputStream and
      StringBufferInputStream, 569
  output and input, 567
  OutputStream methods, 570
  PipedOutputStream and
      PipedInputStream, 579
  PrintStream (see PrintStream)
  serialization and deserialization
      custom, 602
      DataOutputStream and DataInputStream
      classes, 590
      default, 596
      externalization, 604
      virtual machine mechanism, 590
String. See also StringBuffer; StringBuilder
  constructors and methods, 315
  description, 314
  iteration, 318
  literal, 316
  StringBufferInputStream, 568
  String[] getISOCountries() class method, 916
  String[] getISOLanguages() class method, 916
  String[] getArray(String key), 919
  String getString(String key), 919
String representation, 157
Stringtokenizer
  boolean hasMoreElements(), 389
  boolean hasMoreTokens(), 389
description, 388
  int countTokens(), 389
www.it-ebooks.info
loop context, 390
Object nextElement(), 389
parsing or tokenizing, 388
String nextToken(), 389
String nextToken(String delim), 390
String str, 389
String str, String delim, 389
String str, String delim, boolean returnDelims, 389
SVG. See Scalable vector graphics (SVG)
Switch-on-String, 886
Synchronizers
countdown latches
coordinated start, 498–499
java CountDownLatch, 499–500
methods, 497
cyclic barriers
description, 500
methods, 500–501
task into subtasks, decomposing, 501, 504
exchangers
java ExchangerDemo, 508
methods, 505
swap buffers, 505
semaphores
controlling thread access, 511
and fairness, 510
java SemaphoreDemo, 514
methods, 510–511
System.currentTimeMillis() method, 987

T

Threads
description, 324
event-dispatching thread, 924
exception, 333
joining default thread with background, 331
local variables, 352
methods, 326
preemptive scheduling, 329
processors/cores, 329
scheduler, 329
uncaught exception handlers, 334
Timer
boolean isDaemon, 393
cancel() method, 395
description, 391
execution thread, 395
int purge(), 393
String name, 393
String name, boolean isDaemon, 393
task-execution thread, 392
Timer(), 393
void cancel(), 393
void schedule
TimerTask task, Date firstTime,
  long period, 394
TimerTask task, Date time, 393
TimerTask task, long delay, 394
TimerTask task, long delay,
  long period, 394
void scheduleAtFixedRate
TimerTask task, Date firstTime,
  long period, 394
TimerTask task, long delay,
  long period, 395
Timertask
boolean cancel(), 395
description, 391
execution thread, 396
long scheduledExecutionTime(), 396
one-shot execution, 396
repeated execution, 396
void run(), 396
Toolkit class
createImage method, 923
list resource bundles, 923
TreeMap
constructors, 452
definition, 452
demonstration, 452
TreeSet
constructors, 419
description, 419
natural ordering, 420

U, V

Ultimate superclass, 148
Unboxing, 410
  Unicode, internationalization, 914
Uniform Resource Identifiers (URIs)
  absolute, 646
definition, 639
Uniform Resource Identifiers (URIs) (cont.)
fragment, 647
hierarchical, 646
javadoc URIComponents.java, 648
learning about, 648
namespace, 810
normalization, 649
opaque, 646
path, 647
query, 647
registry-based authority, 646
relative, 646
relativization, 651
resolution, 650
scheme-specific-part, 646
server-based authority, 647
Uniform Resource Locator (URL). See also
URLConnection
definition, 639
ListResource's source code, 641
openStream(), 640, 642
protocol prefix, 640
String getProtocol(), 640
String s, 639
Uniform Resource Name (URN), 639
Upcasting
array, 166
description, 164
URI. See Uniform Resource Identifiers (URIs)
URL. See Uniform Resource Locator (URL)
URLConnection
HttpURLConnection, 643
InputStream getInputStream(), 642
OutputStream getOutputStream(), 642
void setDoInput(boolean doInput), 642
void setDoOutput(boolean doOutput), 642
void setRequestProperty(String field, String newValue), 642
URLDecoder
decoding rules, 644
encoded string, 645
String decode(String s, String enc), 644
URLEncoder
encoded string, 645
encoding rules, 643
String encode(String s, String enc), 644
URN. See Uniform Resource Name (URN)
W
WeakHashMap, 465
WeakReference
T r, ReferenceQueue<? super T> rq, 365
WeakReference(T r), 365
X
XML (eXtensible Markup Language)
ASCII character, 806
CDATA section
description, 809
embedded XML document, 809
SVG, 809
character references
color entity reference, 809
description, 808
numeric character reference, 808
commments, 813
custom entity resolver
external entity, 840
LocalRecipeML, 841–842
Recipe Markup Language (RecipeML), 840–841
SAXDemo, 842
declaration, 805–806
description, 803
DOM, parsing and creating (see Document object model (DOM))
elements and attributes
child elements, 807
mixed content, 807
root element, 807
tree structure, 806–807
grammar document
definition, 815
DTD (see Document type definition (DTD))
XML schema, 815
HTML document, 804
namespaces
default, 812–813
Google Chrome, 811
namespace prefix, 810
pair, 810
rules, 814
URI-based container, 810
well-formed documents, 814
XHTML language, 811
non-ASCII characters, 806
processing instructions, 813
SAX, parsing (see Simple API for XML (SAX))
schema
   attribute element, 821
   complex types, 821
   content model, 820
   definition, 820
   element declaration, 821
   ingredient elements, 822
   introductory schema element, 821
   language document, 820
   namespace, 824
   object-oriented approach, 820
   recipe document’s schema, 823
   restriction, 820
valid documents, 814–815
variables and variable resolvers, 873–874
vocabulary documents, 803
XMLPULL V1, parsing
   article.xml file, 858
   event-based, 855
   integer-based event type constants, 857
   JAR file, 855
   org.xmlpull.v1 package, types, 856
   pull parsing, 855
   push parsing, 855
StAX, 855
stream-based, 855
XMLPPDemo, 856
XPath (see XPath)
XPath
   advanced, 868
   and DOM
DOM API, 864, 866
XML-based contacts database, 863
XPath API, 866
attributes, 859
Boolean expressions, 862
compound paths, 860
description, 859
extension functions and function resolvers, 870–872
ingredients, 859
language primer, 859
location path expressions, 859–861
namespace contexts, 868, 870
nodeset, 859
numeric values, functions, 862
predicates, 860–861
strings, functions, 862–863
unknown nodes, wildcards, 860
XPathExpression interface, 868
XPathFactory instance, 867
XSL. See Extensible Stylesheet Language (XSL)
XSLT. See XSL Transformation (XSLT)
XSL Transformation (XSLT)
   demonstrating XSLT API, 878–881
   exploring XSLT API, 875–876, 878
   result tree, 875
Y
Yahoo! Babel Fish, 925
Z
ZIP API, 995
ZIP Archive, 1000
ZipFile vs. ZipInputStream, 1002
Learn Java for Android Development

Jeff Friesen

Apress
To you, dear reader!

I hope this book helps you achieve great success.
Contents

About the Author ................................................................................................................................. xxi
About the Technical Reviewer ............................................................................................................. xxiii
Acknowledgments ............................................................................................................................... xv
Introduction ........................................................................................................................................... xxvii

Chapter 1: Getting Started with Java .................................................................................................. 1
  What Is Java? ........................................................................................................................................... 2
    Java Is a Language .............................................................................................................................. 2
    Java Is a Platform .............................................................................................................................. 4
    Java SE, Java EE, and Java ME ......................................................................................................... 6
  Installing the JDK and Exploring Example Applications .................................................................. 7
    Hello, World! ..................................................................................................................................... 9
    DumpArgs ......................................................................................................................................... 11
    EchoText .......................................................................................................................................... 13
  Installing and Exploring the Eclipse IDE ......................................................................................... 15

Java Meets Android ............................................................................................................................... 20
  What Is Android? .............................................................................................................................. 20
  History of Android ............................................................................................................................ 21
Constructing Objects ................................................................. 92
  Default Constructor .......................................................... 93
  Explicit Constructors ....................................................... 93
  Objects and Applications .................................................. 97
Encapsulating State and Behaviors .......................................... 98
  Representing State via Fields ............................................. 99
  Representing Behaviors via Methods .................................... 107
Hiding Information ................................................................. 119
Initializing Classes and Objects ............................................ 124
  Class Initializers ............................................................ 124
  Instance Initializers ....................................................... 126
  Initialization Order .......................................................... 128
Collecting Garbage ................................................................. 131
Revisiting Arrays .................................................................. 134
Summary ................................................................................ 139

Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces .... 141
Building Class Hierarchies .................................................... 141
  Extending Classes ............................................................ 142
  The Ultimate Superclass .................................................... 148
  Composition ....................................................................... 157
  The Trouble with Implementation Inheritance ....................... 158
Changing Form ...................................................................... 162
  Upcasting and Late Binding ............................................... 163
  Abstract Classes and Abstract Methods ................................ 167
  Downcasting and Runtime Type Identification ....................... 169
  Covariant Return Types ..................................................... 172
Formalizing Class Interfaces .................................................. 174
  Declaring Interfaces .......................................................... 174
  Implementing Interfaces ..................................................... 176
  Extending Interfaces .......................................................... 180
Why Use Interfaces? ............................................................................................................. 181
Summary ................................................................................................................................. 187

Chapter 5: Mastering Advanced Language Features, Part 1 .............................................. 189

Mastering Nested Types ...................................................................................................... 189
  Static Member Classes ........................................................................................................ 189
  Nonstatic Member Classes .................................................................................................. 193
  Anonymous Classes .......................................................................................................... 197
  Local Classes ...................................................................................................................... 200
  Inner Classes and Memory Leaks ...................................................................................... 202
  Interfaces within Classes and Classes within Interfaces .................................................... 203

Mastering Packages .............................................................................................................. 205
  What Are Packages? .......................................................................................................... 206
  The Package Statement ...................................................................................................... 207
  The Import Statement ......................................................................................................... 207
  Searching for Packages and Types .................................................................................... 209
  Playing with Packages ........................................................................................................ 210
  Packages and JAR Files ....................................................................................................... 215

Mastering Static Imports ...................................................................................................... 215

Mastering Exceptions ........................................................................................................... 217
  What Are Exceptions? ......................................................................................................... 218
  Representing Exceptions in Source Code .......................................................................... 218
  Throwing Exceptions ......................................................................................................... 223
  Handling Exceptions ......................................................................................................... 226
  Performing Cleanup .......................................................................................................... 229

Summary ................................................................................................................................. 234

Chapter 6: Mastering Advanced Language Features, Part 2 .............................................. 235

Mastering Assertions ............................................................................................................ 235
  Declaring Assertions ......................................................................................................... 236
  Using Assertions ................................................................................................................ 237
  Avoiding Assertions .......................................................................................................... 243
  Enabling and Disabling Assertions .................................................................................... 244
Mastering Annotations ................................................................. 245

  Discovering Annotations ............................................................ 245
  Declaring Annotation Types and Annotating Source Code .................. 248
  Processing Annotations ............................................................... 253

Mastering Generics ................................................................. 255

  Collections and the Need for Type Safety ..................................... 255
  Generic Types .................................................................................. 258
  Generic Methods ............................................................................. 267
  Arrays and Generics ........................................................................ 271

Mastering Enums ......................................................................... 273

  The Trouble with Traditional Enumerated Types ......................... 273
  The Enum Alternative ..................................................................... 275
  The Enum Class ............................................................................. 280

Summary ....................................................................................... 284

Chapter 7: Exploring the Basic APIs, Part 1 ......................................... 287

Exploring Math ................................................................. 287

  StrictMath and strictfp ................................................................. 294

Exploring Number and Its Children ........................................ 295

  BigDecimal ..................................................................................... 295
  BigInteger ...................................................................................... 301
  Primitive Type Wrapper Classes ...................................................... 305

Exploring String, StringBuffer, and StringBuilder .................. 314

  String ............................................................................................ 314
  StringBuffer and StringBuilder .................................................. 318

Exploring System ....................................................................... 322

Exploring Threads ..................................................................... 324

  Runnable and Thread .................................................................... 325
  Synchronization ............................................................................ 335
  Thread-Local Variables ................................................................. 352

Summary ..................................................................................... 357
Chapter 8: Exploring the Basic APIs, Part 2 ................................................................. 359

Exploring Random ........................................................................................................... 359

Exploring References ....................................................................................................... 362
  Basic Terminology ........................................................................................................... 362
  Reference and ReferenceQueue ...................................................................................... 363
  SoftReference ................................................................................................................ 365
  WeakReference ............................................................................................................. 365
  PhantomReference ....................................................................................................... 366

Exploring Reflection ........................................................................................................ 368
  The Class Entry Point ................................................................................................... 368
  Constructor, Field, and Method .................................................................................... 378
  Package ....................................................................................................................... 383
  Array ............................................................................................................................. 388

Exploring StringTokenizer ............................................................................................. 388

Exploring Timer and TimerTask ..................................................................................... 391
  Timer in Depth .............................................................................................................. 392
  TimerTask in Depth ...................................................................................................... 395

Summary .......................................................................................................................... 399

Chapter 9: Exploring the Collections Framework ...................................................... 401

Exploring Collections Framework Fundamentals ...................................................... 401
  Comparable vs. Comparator ......................................................................................... 402
  Iterable and Collection ............................................................................................... 404

Exploring Lists ................................................................................................................ 411
  ArrayList ...................................................................................................................... 415
  LinkedList ................................................................................................................... 416

Exploring Sets ................................................................................................................ 419
  TreeSet ......................................................................................................................... 419
  HashSet ........................................................................................................................ 420
  EnumSet ....................................................................................................................... 425

Exploring Sorted Sets ................................................................................................... 428

Exploring Navigable Sets .............................................................................................. 435
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploring Queues</td>
<td>438</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>440</td>
</tr>
<tr>
<td>Exploring Deques</td>
<td>443</td>
</tr>
<tr>
<td>ArrayDeque</td>
<td>447</td>
</tr>
<tr>
<td>Exploring Maps</td>
<td>448</td>
</tr>
<tr>
<td>TreeMap</td>
<td>452</td>
</tr>
<tr>
<td>HashMap</td>
<td>453</td>
</tr>
<tr>
<td>IdentityHashMap</td>
<td>463</td>
</tr>
<tr>
<td>WeakHashMap</td>
<td>465</td>
</tr>
<tr>
<td>EnumMap</td>
<td>467</td>
</tr>
<tr>
<td>Exploring Sorted Maps</td>
<td>468</td>
</tr>
<tr>
<td>Exploring Navigable Maps</td>
<td>471</td>
</tr>
<tr>
<td>Exploring the Arrays and Collections Utility APIs</td>
<td>475</td>
</tr>
<tr>
<td>Exploring the Legacy Collection APIs</td>
<td>479</td>
</tr>
<tr>
<td>Summary</td>
<td>486</td>
</tr>
</tbody>
</table>

**Chapter 10: Exploring the Concurrency Utilities** | 487 |

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introducing the Concurrency Utilities</td>
<td>487</td>
</tr>
<tr>
<td>Exploring Executors</td>
<td>488</td>
</tr>
<tr>
<td>Exploring Synchronizers</td>
<td>497</td>
</tr>
<tr>
<td>Countdown Latches</td>
<td>497</td>
</tr>
<tr>
<td>Cyclic Barriers</td>
<td>500</td>
</tr>
<tr>
<td>Exchangers</td>
<td>505</td>
</tr>
<tr>
<td>Semaphores</td>
<td>509</td>
</tr>
<tr>
<td>Exploring the Concurrent Collections</td>
<td>515</td>
</tr>
<tr>
<td>Demonstrating BlockingQueue and ArrayBlockingQueue</td>
<td>516</td>
</tr>
<tr>
<td>Learning More About ConcurrentHashMap</td>
<td>518</td>
</tr>
<tr>
<td>Exploring the Locking Framework</td>
<td>518</td>
</tr>
<tr>
<td>Lock</td>
<td>519</td>
</tr>
<tr>
<td>ReentrantLock</td>
<td>521</td>
</tr>
<tr>
<td>Condition</td>
<td>524</td>
</tr>
</tbody>
</table>
Chapter 12: Accessing Networks ................................................................. 621

Accessing Networks via Sockets ................................................................. 622
  Socket Addresses .................................................................................. 624
  Socket Options ...................................................................................... 625
  Socket and ServerSocket ...................................................................... 626
  DatagramSocket and MulticastSocket .................................................. 632

Accessing Networks via URLs ................................................................. 639
  URL and URLConnection ..................................................................... 639
  URLEncoder and URLDecoder ............................................................... 643
  URI ........................................................................................................ 645

Accessing Network Interfaces and Interface Addresses ......................... 652

Managing Cookies .................................................................................... 657

Summary .................................................................................................. 662

Chapter 13: Migrating to New I/O .............................................................. 665

Working with Buffers ............................................................................... 666
  Buffer and Its Children ........................................................................ 666
  Buffers in Depth ................................................................................... 670

Working with Channels ............................................................................ 681
  Channel and Its Children ..................................................................... 681
  Channels in Depth ................................................................................ 686

Working with Selectors ............................................................................ 723
  Selector Fundamentals .......................................................................... 724
  Selector Demonstration ........................................................................ 728

Working with Regular Expressions .......................................................... 732
  Pattern, PatternSyntaxException, and Matcher ..................................... 732
  Character Classes .................................................................................. 737
  Capturing Groups ................................................................................. 738
  Boundary Matchers and Zero-Length Matches .................................... 739
Chapter 14: Accessing Databases

Introducing Java DB

Java DB Installation and Configuration

Java DB Demos

Java DB Command-Line Tools

Introducing SQLite

Accessing Databases via JDBC

Data Sources, Drivers, and Connections

Exceptions

Statements

Metadata

Summary

Chapter 15: Parsing, Creating, and Transforming XML Documents
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing XML Documents with SAX</td>
<td>824</td>
</tr>
<tr>
<td>Exploring the SAX API</td>
<td>825</td>
</tr>
<tr>
<td>Demonstrating the SAX API</td>
<td>831</td>
</tr>
<tr>
<td>Creating a Custom Entity Resolver</td>
<td>840</td>
</tr>
<tr>
<td>Parsing and Creating XML Documents with DOM</td>
<td>843</td>
</tr>
<tr>
<td>A Tree of Nodes</td>
<td>843</td>
</tr>
<tr>
<td>Exploring the DOM API</td>
<td>845</td>
</tr>
<tr>
<td>Parsing XML Documents with XMLPULL V1</td>
<td>855</td>
</tr>
<tr>
<td>Selecting XML Document Nodes with XPath</td>
<td>859</td>
</tr>
<tr>
<td>XPath Language Primer</td>
<td>859</td>
</tr>
<tr>
<td>XPath and DOM</td>
<td>863</td>
</tr>
<tr>
<td>Advanced XPath</td>
<td>868</td>
</tr>
<tr>
<td>Transforming XML Documents with XSLT</td>
<td>875</td>
</tr>
<tr>
<td>Exploring the XSLT API</td>
<td>875</td>
</tr>
<tr>
<td>Demonstrating the XSLT API</td>
<td>878</td>
</tr>
<tr>
<td>Summary</td>
<td>884</td>
</tr>
<tr>
<td><strong>Chapter 16: Focusing on Odds and Ends</strong></td>
<td>885</td>
</tr>
<tr>
<td>Focusing on Additional Language Features</td>
<td>885</td>
</tr>
<tr>
<td>Numeric Literal Enhancements</td>
<td>886</td>
</tr>
<tr>
<td>Switch-on-String</td>
<td>886</td>
</tr>
<tr>
<td>Diamond Operator</td>
<td>887</td>
</tr>
<tr>
<td>Multicatch</td>
<td>887</td>
</tr>
<tr>
<td>Automatic Resource Management</td>
<td>888</td>
</tr>
<tr>
<td>Focusing on Classloaders</td>
<td>891</td>
</tr>
<tr>
<td>Kinds of Classloaders</td>
<td>891</td>
</tr>
<tr>
<td>Class-Loading Mechanics</td>
<td>892</td>
</tr>
<tr>
<td>Playing with Classloaders</td>
<td>893</td>
</tr>
<tr>
<td>Classloader Difficulties</td>
<td>897</td>
</tr>
<tr>
<td>Classloaders and Resources</td>
<td>899</td>
</tr>
<tr>
<td>Focusing on Console</td>
<td>902</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Focusing on Design Patterns</td>
<td>905</td>
</tr>
<tr>
<td>Understanding Strategy</td>
<td>906</td>
</tr>
<tr>
<td>Implementing Strategy</td>
<td>906</td>
</tr>
<tr>
<td>Focusing on Double Brace Initialization</td>
<td>909</td>
</tr>
<tr>
<td>Focusing on Fluent Interfaces</td>
<td>910</td>
</tr>
<tr>
<td>Focusing on Immutability</td>
<td>911</td>
</tr>
<tr>
<td>Focusing on Internationalization</td>
<td>914</td>
</tr>
<tr>
<td>Locales</td>
<td>915</td>
</tr>
<tr>
<td>Resource Bundles</td>
<td>916</td>
</tr>
<tr>
<td>Break Iterators</td>
<td>935</td>
</tr>
<tr>
<td>Collators</td>
<td>939</td>
</tr>
<tr>
<td>Dates, Time Zones, and Calendars</td>
<td>941</td>
</tr>
<tr>
<td>Formatters</td>
<td>948</td>
</tr>
<tr>
<td>Focusing on Logging</td>
<td>958</td>
</tr>
<tr>
<td>Logging API Overview</td>
<td>959</td>
</tr>
<tr>
<td>A Hierarchy of Loggers</td>
<td>960</td>
</tr>
<tr>
<td>Logging Messages</td>
<td>962</td>
</tr>
<tr>
<td>Filtering LogRecords</td>
<td>968</td>
</tr>
<tr>
<td>Handlers and Formatters</td>
<td>971</td>
</tr>
<tr>
<td>LogManager and Configuration</td>
<td>974</td>
</tr>
<tr>
<td>ErrorManager</td>
<td>977</td>
</tr>
<tr>
<td>Focusing on Preferences</td>
<td>981</td>
</tr>
<tr>
<td>Exploring Preferences</td>
<td>982</td>
</tr>
<tr>
<td>Focusing on Runtime and Process</td>
<td>986</td>
</tr>
<tr>
<td>Focusing on the Java Native Interface</td>
<td>990</td>
</tr>
<tr>
<td>Creating a Hybrid Library</td>
<td>990</td>
</tr>
<tr>
<td>Testing the Hybrid Library</td>
<td>994</td>
</tr>
<tr>
<td>Focusing on ZIP and JAR</td>
<td>995</td>
</tr>
<tr>
<td>Focusing on the ZIP API</td>
<td>995</td>
</tr>
<tr>
<td>Focusing on the JAR API</td>
<td>1003</td>
</tr>
<tr>
<td>Summary</td>
<td>1010</td>
</tr>
</tbody>
</table>
About the Author

Jeff Friesen is a freelance tutor, author, and software developer with an emphasis on Java, Android, and HTML5. In addition to writing this book and its two predecessors, Jeff has written numerous articles on Java and other technologies for JavaWorld (www.javaworld.com), informIT (www.informit.com), java.net, SitePoint (www.sitepoint.com), and others. Jeff can be contacted via his web site at tutortutor.ca.
About the Technical Reviewer

Chád Darby is an author, instructor, and speaker in the Java development world. As a recognized authority on Java applications and architectures, he has presented technical sessions at software development conferences worldwide (U.S., U.K., India, Russia, and Australia). In his 15 years as a professional software architect, he’s had the opportunity to work for Blue Cross/Blue Shield, Merck, Boeing, Red Hat, and a handful of startup companies.

Chád is a contributing author to several Java books, including *Professional Java E-Commerce* (Wrox Press), *Beginning Java Networking* (Wrox Press), and *XML and Web Services Unleashed* (Sams Publishing). Chád has Java certifications from Sun Microsystems and IBM. He holds a B.S. in Computer Science from Carnegie Mellon University.

Visit Chád’s blog at [www.luv2code.com](http://www.luv2code.com) to view his free video tutorials on Java. You can also follow him on Twitter at [@darbyluvs2code](https://twitter.com/darbyluvs2code).
Acknowledgments

I would like to thank Steve Anglin for contacting me to write this book, Jill Balzano and Anamika Panchoo for guiding me through the various aspects of this project, Tom Welsh and Gary Schwartz for helping me with the development of my chapters, and Chad Darby for his diligence in catching various flaws that would otherwise have made it into this book.