Part 1: Introducing Java

Chapter 1. Introduction

Contents:

Welcome to Java. Since its introduction in late 1995, the Java language and platform have taken the programming world by storm. This chapter begins by explaining what Java is and why it has become so popular. Then, as a tutorial introduction to the language, it walks you through a simple Java program you can type in, compile, and run.

1.1. What Is Java?

In discussing Java, it is important to distinguish between the Java programming language, the Java Virtual Machine, and the Java platform. The Java programming language is the language in which Java applications (including applets, servlets, and JavaBeans components) are written. When a Java program is compiled, it is converted to byte codes that are the portable machine language of a CPU architecture known as the Java Virtual Machine (also called the Java VM or JVM). The JVM can be implemented directly in hardware, but it is usually implemented in the form of a software program that interprets and executes byte codes.

The Java platform is distinct from both the Java language and Java VM. The Java platform is the predefined set of Java classes that exist on every Java installation; these classes are available for use by all Java programs. The Java platform is also sometimes referred to as the Java runtime environment or the core Java APIs (application programming interfaces). The Java platform can be extended with optional standard extensions. These extension APIs exist in some Java installations, but are not guaranteed to exist in all installations.

1.1.1. The Java Programming Language

The Java programming language is a state-of-the-art, object-oriented language that has a syntax similar to that of C. The language designers strove to make the Java language powerful, but, at the same time,

they tried to avoid the overly complex features that have bogged down other object-oriented languages, such as C++. By keeping the language simple, the designers also made it easier for programmers to write robust, bug-free code. As a result of its elegant design and next-generation features, the Java language has proved wildly popular with programmers, who typically find it a pleasure to work with Java after struggling with more difficult, less powerful languages.

1.1.2. The Java Virtual Machine

The Java Virtual Machine, or Java interpreter, is the crucial piece of every Java installation. By design, Java programs are portable, but they are only portable to platforms to which a Java interpreter has been ported. Sun ships VM implementations for its own Solaris operating system and for Microsoft Windows (95/98/NT) platforms. Many other vendors, including Apple and various Unix vendors, provide Java interpreters for their platforms. There is a freely available port of Sun's VM for Linux platforms, and there are also other third-party VM implementations available. The Java VM is not only for desktop systems, however. It has been ported to set-top boxes, and versions are even available for hand-held devices that run Windows CE and PalmOS.

Although interpreters are not typically considered high-performance systems, Java VM performance is remarkably good and has been improving steadily. Of particular note is a VM technology called just-in-time (JIT) compilation, whereby Java byte codes are converted on-the-fly into native-platform machine language, boosting execution speed for code that is run repeatedly. Sun's new Hotspot technology is a particularly good implementation of JIT compilation.

1.1.3. The Java Platform

The Java platform is just as important as the Java programming language and the Java Virtual Machine. All programs written in the Java language rely on the set of predefined classes[1] that comprise the Java platform. Java classes are organized into related groups known as *packages*. The Java platform defines packages for functionality such as input/output, networking, graphics, user-interface creation, security, and much more.

[1]A *class* is a module of Java code that defines a data structure and a set of methods (also called procedures, functions, or subroutines) that operate on that data.

The Java 1.2 release was a major milestone for the Java platform. This release almost tripled the number of classes in the platform and introduced significant new functionality. In recognition of this, Sun named the new version the Java 2 Platform. This is a trademarked name created for marketing purposes; it serves to emphasize how much Java has grown since its first release. However, most programmers refer to the Java platform by its official version number, which, at the time of this writing, is 1.2.[2]

[2]Although there is currently a beta release of Java 1.3 available

It is important to understand what is meant by the term platform. To a computer programmer, a platform is defined by the APIs he or she can rely on when writing programs. These APIs are usually defined by the operating system of the target computer. Thus, a programmer writing a program to run under Microsoft Windows must use a different set of APIs than a programmer writing the same program for the Macintosh or for a Unix-based system. In this respect, Windows, Macintosh, and Unix are three distinct platforms.

Java is not an operating system.[3] Nevertheless, the Java platform--particularly the Java 2 Platform--provides APIs with a comparable breadth and depth to those defined by an operating system. With the Java 2 Platform, you can write applications in Java without sacrificing the advanced features available to programmers writing native applications targeted at a particular underlying operating system. An application written on the Java platform runs on any operating system that supports the Java platform. This means you do not have to create distinct Windows, Macintosh, and Unix versions of your programs, for example. A single Java program runs on all these operating systems, which explains why "Write once, run anywhere" is Sun's motto for Java.

[3]There is a Java-based operating system, however; it is known as JavaOS.

It also explains why companies like Microsoft might feel threatened by Java. The Java platform is not an operating system, but for programmers, it is an alternative development target and a very popular one at that. The Java platform reduces programmers' reliance on the underlying operating system, and, by allowing programs to run on top of any operating system, it increases end users' freedom to choose an operating system.

1.1.4. Versions of Java

As of this writing, there have been four major versions of Java. They are:

Java 1.0

This was the first public version of Java. It contained 212 classes organized in 8 packages. There is a large installed base of web browsers that run this version of Java, so this version is still in use for writing simple applets--Java programs that are included in web pages. (See *Java Foundation Classes in a Nutshell* (O'Reilly) for a discussion of applets.)

Java 1.1

This release of Java doubled the size of the Java platform to 504 classes in 23 packages. It introduced inner classes, an important change to the Java language itself, and included significant performance improvements in the Java VM. This version of Java is out of date, but is still in use on systems that do not yet have a stable port of Java 1.2.

Java 1.2

This is the latest and greatest significant release of Java; it tripled the size of the Java platform to 1520 classes in 59 packages. Because of the many new features included in this release, the platform was renamed and is now called the Java 2 Platform.

Java 1.3 (beta)

This release includes minor corrections and updates to the Java platform, but does not include major changes or significant new functionality.

In addition, Sun has instituted a process for proposing and developing standard extensions to the Java platform. In the future, most new functionality is expected to take the form of a standard extension, rather than be a required part of every Java installation.

In order to work with Java 1.0 or Java 1.1, you have to obtain the Java Development Kit (JDK) for that release. As of Java 1.2, the JDK has been renamed and is now called a Software Development Kit (SDK), so we have the Java 2 SDK or, more precisely, the Java 2 SDK, Standard Edition, Version 1.2 (or Version 1.3 beta). Despite the new name, many programmers still refer to the development kit as the JDK.

Don't confuse the JDK (or SDK) with the Java Runtime Environment (JRE). The JRE contains everything you need to run Java programs, but does not contain the tools you need to develop Java programs (i.e., the compiler). You should also be aware of the Java Plug-in, a version of the Java 1.2 (and 1.3) JRE that is designed to be integrated into the Netscape Navigator and Microsoft Internet Explorer web browsers.

In addition to evolving the Java platform over time, Sun is also trying to produce different versions of the platform for different uses. The Standard Edition is the only version currently available, but Sun is also working on the Java 2 Platform, Enterprise Edition (J2EE), for enterprise developers and the Java 2 Platform, Micro Edition, for consumer electronic systems, like handheld PDAs and cellular telephones.

1.2. Key Benefits of Java

Why use Java at all? Is it worth learning a new language and a new platform? This section explores some of the key benefits of Java.

1.2.1. Write Once, Run Anywhere

Sun identifies "Write once, run anywhere" as the core value proposition of the Java platform. Translated from business jargon, this means that the most important promise of Java technology is that you only have to write your application once--for the Java platform--and then you'll be able to run it *anywhere*.

Anywhere, that is, that supports the Java platform. Fortunately, Java support is becoming ubiquitous. It is integrated, or being integrated, into practically all major operating systems. It is built into the popular web browsers, which places it on virtually every Internet-connected PC in the world. It is even being built into consumer electronic devices, such as television set-top boxes, PDAs, and cell phones.

1.2.2. Security

Another key benefit of Java is its security features. Both the language and the platform were designed from the ground up with security in mind. The Java platform allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: it cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.

The Java 2 Platform takes the security model a step further. It makes security levels and restrictions highly configurable and extends them beyond applets. As of Java 1.2, any Java code, whether it is an applet, a servlet, a JavaBeans component, or a complete Java application, can be run with restricted permissions that prevent it from doing harm to the host system.

The security features of the Java language and platform have been subjected to intense scrutiny by security experts around the world. Security-related bugs, some of them potentially serious, have been found and promptly fixed. Because of the security promises Java makes, it is big news when a new security bug is found. Remember, however, that no other mainstream platform can make security guarantees nearly as strong as those Java makes. If Java's security is not yet perfect, it has been proven strong enough for practical day-to-day use and is certainly better than any of the alternatives.

1.2.3. Network-centric Programming

Sun's corporate motto has always been "The network is the computer." The designers of the Java platform believed in the importance of networking and designed the Java platform to be network-centric. From a programmer's point of view, Java makes it unbelievably easy to work with resources across a network and to create network-based applications using client/server or multitier architectures. This means that Java programmers have a serious head start in the emerging network economy.

1.2.4. Dynamic, Extensible Programs

Java is both dynamic and extensible. Java code is organized in modular object-oriented units called *classes*. Classes are stored in separate files and are loaded into the Java interpreter only when needed. This means that an application can decide as it is running what classes it needs and can load them when it needs them. It also means that a program can dynamically extend itself by loading the classes it needs to expand its functionality.

The network-centric design of the Java platform means that a Java application can dynamically extend itself by loading new classes over a network. An application that takes advantage of these features ceases to be a monolithic block of code. Instead, it becomes an interacting collection of independent software components. Thus, Java enables a powerful new metaphor of application design and development.

1.2.5. Internationalization

The Java language and the Java platform were designed from the start with the rest of the world in mind. Java is the only commonly used programming language that has internationalization features at its very core, rather than tacked on as an afterthought. While most programming languages use 8-bit characters that represent only the alphabets of English and Western European languages, Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world. Java's internationalization features are not restricted to just low-level character representation, however. The features permeate the Java platform, making it easier to write internationalized programs with Java than it is with any other environment.

1.2.6. Performance

As I described earlier, Java programs are compiled to a portable intermediate form known as byte codes, rather than to native machine-language instructions. The Java Virtual Machine runs a Java program by interpreting these portable byte-code instructions. This architecture means that Java programs are faster than programs or scripts written in purely interpreted languages, but they are typically slower than C and C++ programs compiled to native machine language. Keep in mind, however, that although Java programs are compiled to byte code, not all of the Java platform is implemented with interpreted byte

codes. For efficiency, computationally intensive portions of the Java platform--such as the string-manipulation methods--are implemented using native machine code.

Although early releases of Java suffered from performance problems, the speed of the Java VM has improved dramatically with each new release. The VM has been highly tuned and optimized in many significant ways. Furthermore, many implementations include a just-in-time compiler, which converts Java byte codes to native machine instructions on the fly. Using sophisticated JIT compilers, Java programs can execute at speeds comparable to the speeds of native C and C++ applications.

Java is a portable, interpreted language; Java programs run almost as fast as native, non-portable C and C++ programs. Performance used to be an issue that made some programmers avoid using Java. Now, with the improvements made in Java 1.2, performance issues should no longer keep anyone away. In fact, the winning combination of performance plus portability is a unique feature no other language can offer.

1.2.7. Programmer Efficiency and Time-to-Market

The final, and perhaps most important, reason to use Java is that programmers like it. Java is an elegant language combined with a powerful and well-designed set of APIs. Programmers enjoy programming in Java and are usually amazed at how quickly they can get results with it. Studies have consistently shown that switching to Java increases programmer efficiency. Because Java is a simple and elegant language with a well-designed, intuitive set of APIs, programmers write better code with fewer bugs than for other platforms, again reducing development time.

Chapter 1: Introduction

1.3. An Example Program

Example 1-1 shows a Java program to compute factorials.[4] The numbers at the beginning of each line are not part of the program; they are there for ease of reference when we dissect the program line-by-line.

> [4]The factorial of an integer is the product of the number and all positive integers less than the number. So, for example, the factorial of 4, which is also written 4!, is 4 times 3 times 2 times 1, or 24. By definition, 0! is 1.

Example 1-1. Factorial.java: A Program to Compute Factorials

```
 1 /**
 2  * This program computes the factorial of a number
 3  */
 4 public class Factorial {                        // Define a class
 5   public static void main(String[] args) { // The program starts here
 6     int input = Integer.parseInt(args[0]); // Get the user's input
 7     double result = factorial(input);      // Compute the factorial
 8     System.out.println(result);            // Print out the result
 9   }                                        // The main() method ends here
10
11   public static double factorial(int x) {  // This method computes x!
12     if (x < 0)                             // Check for bad input
13       return 0.0;                          //   if bad, return 0
14     double fact = 1.0;                     // Begin with an initial value
15     while(x > 1) {                         // Loop until x equals 1
16       fact = fact * x;                     //   multiply by x each time
17       x = x - 1;                           //   and then decrement x
18     }                                      // Jump back to start of loop
19     return fact;                           // Return the result
20   }                                        // factorial() ends here
21 }                                          // The class ends here
```

1.3.1. Compiling and Running the Program

Before we look at how the program works, we must first discuss how to run it. In order to compile and run the program, you need a Java software development kit (SDK) of some sort. Sun Microsystems created the Java language and ships a free Java SDK for its Solaris operating system and for Microsoft Windows (95/98/NT) platforms. At the time of this writing, the current version of Sun's SDK is entitled Java 2 SDK, Standard Edition, Version 1.2.2 and is available for download from http://java.sun.com/products/jdk/1.2/ (Sun's Java SDK is still often called the JDK, even internally). Be sure to get the SDK and not the Java Runtime Environment. The JRE enables you to run existing Java programs, but not

to write your own.

Sun supports its SDK only on Solaris and Windows platforms. Many other companies have licensed and ported the SDK to their platforms, however. Contact your operating-system vendor to find if a version of the Java SDK is available for your system. Linux users should visit http://www.blackdown.org/.

The Sun SDK is not the only Java programming environment you can use. Companies such as Borland, Inprise, Metrowerks, Oracle, Sybase, and Symantec offer commercial products that enable you to write Java programs. This book assumes that you are using Sun's SDK. If you are using a product from some other vendor, be sure to read that vendor's documentation to learn how to compile and run a simple program, like that shown in Example 1-1.

Once you have a Java programming environment installed, the first step towards running our program is to type it in. Using your favorite text editor, enter the program as it is shown in Example 1-1. Omit the line numbers, as they are just there for reference. Note that Java is a case-sensitive language, so you must type lowercase letters in lowercase and uppercase letters in uppercase. You'll notice that many of the lines of this program end with semicolons. It is a common mistake to forget these characters, but the program won't work without them, so be careful! If you are not a fast typist, you can omit everything from `//` to the end of a line. Those are *comments* ; they are there for your benefit and are ignored by Java.[5]

> [5]I recommend that you type this example in by hand, to get a feel for the language. If you *really* don't want to, however, you can download this, and all examples in the book, from http://www.oreilly.com/catalog/javanut3/.

When writing Java programs, you should use a text editor that saves files in plain-text format, not a word processor that supports fonts and formatting and saves files in a proprietary format. My favorite text editor on Unix systems is *emacs*. If you use a Windows system, you might use *Notepad* or *WordPad*, if you don't have a more specialized programmer's editor. If you are using a commercial Java programming environment, it probably includes an appropriate text editor; read the documentation that came with the product. When you are done entering the program, save it in a file named *Factorial.java*. This is important; the program will not work if you save it by any other name.

After writing a program like this one, the next step is to compile it. With Sun's SDK, the Java compiler is known as *javac*. *javac* is a command-line tool, so you can only use it from a terminal window, such as an MS-DOS window on a Windows system or an *xterm* window on a Unix system. Compile the program by typing the following command line:[6]

> [6]The "C:\>" characters represent the command-line prompt; *don't* type these characters yourself.

```
C:\> javac Factorial.java
```

If this command prints any error messages, you probably got something wrong when you typed in the program. If it does not print any error messages, however, the compilation has succeeded, and *javac* creates a file called *Factorial.class*. This is the compiled version of the program.

Once you have compiled a Java program, you must still run it. Unlike some other languages, Java programs are not compiled into native machine language, so they cannot be executed directly by the system. Instead, they are run by another program known as the Java interpreter. In Sun's SDK, the interpreter is a command-line program named, appropriately enough, *java*. To run the factorial program, type:

```
C:\> java Factorial 4
```

java is the command to run the Java interpreter, *Factorial* is the name of the Java program we want the interpreter to run, and *4* is the input data--the number we want the interpreter to compute the factorial of. The program prints a single line of output, telling us that the factorial of 4 is 24:

```
C:\> java Factorial 4
24.0
```

Congratulations! You've just written, compiled, and run your first Java program. Try running it again to compute the factorials of some other numbers.

1.3.2. Analyzing the Program

Now that you have run the factorial program, let's analyze it line by line, to see what makes a Java program tick.

1.3.2.1. Comments

The first three lines of the program are a comment. Java ignores them, but they tell a human programmer what the program does. A comment begins with the characters /* and ends with the characters */. Any amount of text, including multiple lines of text, may appear between these characters. Java also supports another type of comment, which you can see in lines 4 through 21. If the characters // appear in a Java program, Java ignores those characters and any other text that appears between those characters and the end of the line.

1.3.2.2. Defining a class

Line 4 is the beginning of the program. It says that we are defining a class named Factorial. This explains why the program had to be stored in a file named *Factorial.java*. That filename indicates that the file contains Java source code for a class named Factorial. The word public is a *modifier* ; it says that the class is publicly available and that anyone may use it. The open curly-brace character ({) marks the beginning of the body of the class, which extends all the way to line 21, where we find the matching close curly-brace character (}). The program contains a number of pairs of curly braces; the lines are indented to show the nesting within these braces.

A class is the fundamental unit of program structure in Java, so it is not surprising that the first line of our program declares a class. All Java programs are classes, although some programs use many classes instead of just one. Java is an object-oriented programming language, and classes are a fundamental part of the object-oriented paradigm. Each class defines a unique kind of object. Example 1-1 is not really an object-oriented program, however, so I'm not going to go into detail about classes and objects here. That is the topic of Chapter 3, "Object-Oriented Programming in Java". For now, all you need to understand is that a class defines a set of interacting *members*. Those members may be fields, methods, or other classes. The Factorial class contains two members, both of which are methods. They are described in upcoming sections.

1.3.2.3. Defining a method

Line 5 begins the definition of a *method* of our Factorial class. A method is a named chunk of Java code. A Java program can call, or *invoke*, a method to execute the code in it. If you have programmed in other languages, you have

probably seen methods before, but they may have been called functions, procedures, or subroutines. The interesting thing about methods is that they have *parameters* and *return values*. When you call a method, you pass it some data you want it to operate on, and it returns a result to you. A method is like an algebraic function:

```
y = f(x)
```

Here, the mathematical function f performs some computation on the value represented by x and returns a value, which we represent by y.

To return to line 5, the `public` and `static` keywords are modifiers. `public` means the method is publicly accessible; anyone can use it. The meaning of the `static` modifier is not important here; it is explained in Chapter 3, "Object-Oriented Programming in Java". The `void` keyword specifies the return value of the method. In this case, it specifies that this method does not have a return value.

The word `main` is the name of the method. `main` is a special name. When you run the Java interpreter, it reads in the class you specify, then looks for a method named `main()`.[7] When the interpreter finds this method, it starts running the program at that method. When the `main()` method finishes, the program is done, and the Java interpreter exits. In other words, the `main()` method is the main entry point into a Java program. It is not actually sufficient for a method to be named `main()`, however. The method must be declared `public static void` exactly as shown in line 5. In fact, the only part of line 5 you can change is the word `args`, which you can replace with any word you want. You'll be using this line in all of your Java programs, so go ahead and commit it to memory now![8]

[7]By convention, when this book refers to a method, it follows the name of the method by a pair of parentheses. As you'll see, parentheses are an important part of method syntax, and they serve here to keep method names distinct from the names of classes, fields, variables, and so on.

[8]All Java programs that are run directly by the Java interpreter must have a `main()` method. Programs of this sort are often called *applications*. It is possible to write programs that are not run directly by the interpreter, but are dynamically loaded into some other already running Java program. Examples are *applets*, which are programs run by a web browser, and *servlets*, which are programs run by a web server. Applets are discussed in *Java Foundation Classes in a Nutshell* (O'Reilly), while servlets are discussed in *Java Enterprise in a Nutshell* (O'Reilly). In this book, we consider only applications.

Following the name of the `main()` method is a list of method parameters, contained in parentheses. This `main()` method has only a single parameter. `String[]` specifies the type of the parameter, which is an array of strings (i.e., a numbered list of strings of text). `args` specifies the name of the parameter. In the algebraic equation `f(x)`, `x` is simply a way of referring to an unknown value. `args` serves the same purpose for the `main()` method. As we'll see, the name `args` is used in the body of the method to refer to the unknown value that is passed to the method.

As I've just explained, the `main()` method is a special one that is called by the Java interpreter when it starts running a Java class (program). When you invoke the Java interpreter like this:

```
C:\> java Factorial 4
```

the string "4" is passed to the `main()` method as the value of the parameter named `args`. More precisely, an array of strings containing only one entry, "4", is passed to `main()`. If we invoke the program like this:

```
C:\> java Factorial 4 3 2 1
```

then an array of four strings, "4", "3", "2", and "1", are passed to the `main()` method as the value of the parameter named `args`. Our program looks only at the first string in the array, so the other strings are ignored.

Finally, the last thing on line 5 is an open curly brace. This marks the beginning of the body of the `main()` method, which continues until the matching close curly brace on line 9. Methods are composed of *statements*, which the Java interpreter executes in sequential order. In this case, lines 6, 7, and 8 are three statements that compose the body of the `main()` method. Each statement ends with a semicolon to separate it from the next. This is an important part of Java syntax; beginning programmers often forget the semicolons.

1.3.2.4. Declaring a variable and parsing input

The first statement of the `main()` method, line 6, declares a variable and assigns a value to it. In any programming language, a *variable* is simply a symbolic name for a value. Think back to algebra class again:

$$c^2 = a^2 + b^2$$

The letters `a`, `b`, and `c` are names we use to refer to unknown values. They make this formula (the Pythagorean theorem) a general one that applies to arbitrary values of `a`, `b`, and `c`, not just a specific set like:

$$5^2 = 4^2 + 3^2$$

A variable in a Java program is exactly the same thing: it is a name we use to refer to a value. More precisely, a variable is a name that refers to a storage space for a value. We often say that a variable holds a value.

Line 6 begins with the words `int input`. This declares a variable named `input` and specifies that the variable has the type `int`; that is, it is an integer. Java can work with several different types of values, including integers, real or floating-point numbers, characters (e.g., letters, digits), and strings. Java is a *strongly typed* language, which means that all variables must have a type specified and can only refer to values of that type. Our `input` variable always refers to an integer; it cannot refer to a floating point number or a string. Method parameters are also typed. Recall that the `args` parameter had a type of `String[]`.

Continuing with line 6, the variable declaration `int input` is followed by the `=` character. This is the assignment operator in Java; it sets the value of a variable. When reading Java code, don't read `=` as "equals," but instead read it as "is assigned the value." As we'll see in [Chapter 2, "Java Syntax from the Ground Up"](#), there is a different operator for "equals."

The value being assigned to our `input` variable is `Integer.parseInt(args[0])`. This is a method invocation. This first statement of the `main()` method invokes another method whose name is `Integer.parseInt()`. As you might guess, this method "parses" an integer; that is, it converts a string representation of an integer, such as "4", to the integer itself. The `Integer.parseInt()` method is not part of the Java language, but it is a core part of the Java API or Application Programming Interface. Every Java program can use the powerful set of classes and methods defined by this core API. The second half of this book is a quick-reference that documents that core API.

When you call a method, you pass values (called *arguments*) that are assigned to the corresponding parameters defined by the method, and the method returns a value. The argument passed to `Integer.parseInt()` is `args[0]`. Recall

that `args` is the name of the parameter for `main()`; it specifies an array (or list) of strings. The elements of an array are numbered sequentially, and the first one is always numbered 0. We only care about the first string in the `args` array, so we use the expression `args[0]` to refer to that string. Thus, when we invoke the program as shown earlier, line 6 takes the first string specified after the name of the class, "4", and passes it to the method named `Integer.parseInt()`. This method converts the string to the corresponding integer and returns the integer as its return value. Finally, this returned integer is assigned to the variable named `input`.

1.3.2.5. Computing the result

The statement on line 7 is a lot like the statement on line 6. It declares a variable and assigns a value to it. The value assigned to the variable is computed by invoking a method. The variable is named `result`, and it has a type of `double`. `double` means a double-precision floating-point number. The variable is assigned a value that is computed by the `factorial()` method. The `factorial()` method, however, is not part of the standard Java API. Instead, it is defined as part of our program, by lines 11 through 19. The argument passed to `factorial()` is the value referred to by the `input` variable, which was computed on line 6. We'll consider the body of the `factorial()` method shortly, but you can surmise from its name that this method takes an input value, computes the factorial of that value, and returns the result.

1.3.2.6. Displaying output

Line 8 simply calls a method named `System.out.println()`. This commonly used method is part of the core Java API; it causes the Java interpreter to print out a value. In this case, the value that it prints is the value referred to by the variable named `result`. This is the result of our factorial computation. If the `input` variable holds the value 4, the `result` variable holds the value 24, and this line prints out that value.

The `System.out.println()` method does not have a return value, so there is no variable declaration or = assignment operator in this statement, since there is no value to assign to anything. Another way to say this is that, like the `main()` method of line 5, `System.out.println()` is declared `void`.

1.3.2.7. The end of a method

Line 9 contains only a single character, `}`. This marks the end of the method. When the Java interpreter gets here, it is done executing the `main()` method, so it stops running. The end of the `main()` method is also the end of the *variable scope* for the `input` and `result` variables declared within `main()` and for the `args` parameter of `main()`. These variable and parameter names have meaning only within the `main()` method and cannot be used elsewhere in the program, unless other parts of the program declare different variables or parameters that happen to have the same name.

1.3.2.8. Blank lines

Line 10 is a blank line. You can insert blank lines, spaces, and tabs anywhere in a program, and you should use them liberally to make the program readable. A blank line appears here to separate the `main()` method from the `factorial()` method that begins on line 11. You'll notice that the program also uses spaces and tabs to indent the various lines of code. This kind of indentation is optional; it emphasizes the structure of the program and greatly enhances the readability of the code.

1.3.2.9. Another method

Line 11 begins the definition of the `factorial()` method that was used by the `main()` method. Compare this line to line 5 to note its similarities and differences. The `factorial()` method has the same `public` and `static` modifiers. It takes a single integer parameter, which we call `x`. Unlike the `main()` method, which had no return value (`void`), `factorial()` returns a value of type `double`. The open curly brace marks the beginning of the method body, which continues past the nested braces on lines 15 and 18 to line 20, where the matching close curly brace is found. The body of the `factorial()` method, like the body of the `main()` method, is composed of statements, which are found on lines 12 through 19.

1.3.2.10. Checking for valid input

In the `main()` method, we saw variable declarations, assignments, and method invocations. The statement on line 12 is different. It is an `if` statement, which executes another statement conditionally. We saw earlier that the Java interpreter executes the three statements of the `main()` method one after another. It always executes them in exactly that way, in exactly that order. An `if` statement is a flow-control statement; it can affect the way the interpreter runs a program.

The `if` keyword is followed by a parenthesized expression and a statement. The Java interpreter first evaluates the expression. If it is `true`, the interpreter executes the statement. If the expression is `false`, however, the interpreter skips the statement and goes to the next one. The condition for the `if` statement on line 12 is `x < 0`. It checks whether the value passed to the `factorial()` method is less than zero. If it is, this expression is `true`, and the statement on line 13 is executed. Line 12 does not end with a semicolon because the statement on line 13 is part of the `if` statement. Semicolons are required only at the end of a statement.

Line 13 is a `return` statement. It says that the return value of the `factorial()` method is 0.0. `return` is also a flow-control statement. When the Java interpreter sees a `return`, it stops executing the current method and returns the specified value immediately. A `return` statement can stand alone, but in this case, the `return` statement is part of the `if` statement on line 12. The indentation of line 13 helps emphasize this fact. (Java ignores this indentation, but it is very helpful for humans who read Java code!) Line 13 is executed only if the expression on line 12 is `true`.

Before we move on, we should pull back a bit and talk about why lines 12 and 13 are necessary in the first place. It is an error to try to compute a factorial for a negative number, so these lines make sure that the input value `x` is valid. If it is not valid, they cause `factorial()` to return a consistent invalid result, 0.0.

1.3.2.11. An important variable

Line 14 is another variable declaration; it declares a variable named `fact` of type `double` and assigns it an initial value of 1.0. This variable holds the value of the factorial as we compute it in the statements that follow. In Java, variables can be declared anywhere; they are not restricted to the beginning of a method or block of code.

1.3.2.12. Looping and computing the factorial

Line 15 introduces another type of statement: the `while` loop. Like an `if` statement, a `while` statement consists of a parenthesized expression and a statement. When the Java interpreter sees a `while` statement, it evaluates the associated expression. If that expression is `true`, the interpreter executes the statement. The interpreter repeats this process, evaluating the expression and executing the statement if the expression is `true`, until the expression evaluates to `false`. The expression on line 15 is `x > 1`, so the `while` statement loops *while* the parameter `x` holds a value that is greater than 1. Another way to say this is that the loop continues *until* `x` holds a value less than or equal to 1. We can assume from this expression that if the loop is ever going to terminate, the value of `x` must somehow be modified by the

statement that the loop executes.

The major difference between the `if` statement on lines 12-13 and the `while` loop on lines 15-18 is that the statement associated with the `while` loop is a *compound statement*. A compound statement is zero or more statements grouped between curly braces. The `while` keyword on line 15 is followed by an expression in parentheses and then by an open curly brace. This means that the body of the loop consists of all statements between that opening brace and the closing brace on line 18. Earlier in the chapter, I said that all Java statements end with semicolons. This rule does not apply to compound statements, however, as you can see by the lack of a semicolon at the end of line 18. The statements inside the compound statement (lines 16 and 17) do end with semicolons, of course.

The body of the `while` loop consists of the statements on line 16 and 17. Line 16 multiplies the value of `fact` by the value of `x` and stores the result back into `fact`. Line 17 is similar. It subtracts 1 from the value of `x` and stores the result back into `x`. The `*` character on line 16 is important: it is the multiplication *operator*. And, as you can probably guess, the `-` on line 17 is the subtraction operator. An operator is a key part of Java syntax: it performs a computation on one or two *operands* to produce a new value. Operands and operators combine to form *expressions*, such as `fact * x` or `x - 1`. We've seen other operators in the program. Line 15, for example, uses the greater-than operator (`>`) in the expression `x > 1`, which compares the value of the variable `x` to 1. The value of this expression is a boolean truth value--either `true` or `false`, depending on the result of the comparison.

To understand this `while` loop, it is helpful to think like the Java interpreter. Suppose we are trying to compute the factorial of 4. Before the loop starts, `fact` is 1.0, and `x` is 4. After the body of the loop has been executed once--after the first *iteration*--`fact` is 4.0, and `x` is 3. After the second iteration, `fact` is 12.0, and `x` is 2. After the third iteration, `fact` is 24.0, and `x` is 1. When the interpreter tests the loop condition after the third iteration, it finds that `x > 1` is no longer true, so it stops running the loop, and the program resumes at line 19.

1.3.2.13. Returning the result

Line 19 is another `return` statement, like the one we saw on line 13. This one does not return a constant value like 0.0, but instead returns the value of the `fact` variable. If the value of `x` passed into the `factorial()` function is 4, then, as we saw earlier, the value of `fact` is 24.0, so this is the value returned. Recall that the `factorial()` method was invoked on line 7 of the program. When this `return` statement is executed, control returns to line 7, where the return value is assigned to the variable named `result`.

1.3.3. Exceptions

If you've made it all the way through the line-by-line analysis of Example 1-1, you are well on your way to understanding the basics of the Java language.[9] It is a simple but nontrivial program that illustrates many of the features of Java. There is one more important feature of Java programming I want to introduce, but it is one that does not appear in the program listing itself. Recall that the program computes the factorial of the number you specify on the command line. What happens if you run the program without specifying a number?

[9] If you didn't understood all the details of this factorial program, don't worry. We'll cover the details of the Java language a lot more thoroughly in Chapter 2, "Java Syntax from the Ground Up" and Chapter 3, "Object-Oriented Programming in Java". However, if you feel like you didn't understand any of the line-by-line analysis, you may also find that the upcoming chapters are over your head. In that case, you should probably go elsewhere to learn the basics of the Java language and return to this book to solidify your understanding, and, of course, to use as a reference. One resource you may find useful in learning the

language is Sun's online Java tutorial, available at http://java.sun.com/docs/books/tutorial/.

```
C:\> java Factorial
java.lang.ArrayIndexOutOfBoundsException: 0
        at Factorial.main(Factorial.java:6)
C:\>
```

And what happens if you specify a value that is not a number?

```
C:\> java Factorial ten
java.lang.NumberFormatException: ten
        at java.lang.Integer.parseInt(Integer.java)
        at java.lang.Integer.parseInt(Integer.java)
        at Factorial.main(Factorial.java:6)
C:\>
```

In both cases, an error occurs or, in Java terminology, an *exception* is thrown. When an exception is thrown, the Java interpreter prints out a message that explains what type of exception it was and where it occurred (both exceptions above occurred on line 6•). In the first case, the exception is thrown because there are no strings in the args list, meaning we asked for a nonexistent string with args[0]. In the second case, the exception is thrown because Integer.parseInt() cannot convert the string "ten" to a number. We'll see more about exceptions in Chapter 2, "Java Syntax from the Ground Up" and learn how to handle them gracefully as they occur.

Chapter 2. Java Syntax from the Ground Up

Contents:

This chapter is a terse but comprehensive introduction to Java syntax. It is written primarily for readers who are new to the language, but have at least some previous programming experience. Determined novices with no prior programming experience may also find it useful. If you already know Java, you should find it a useful language reference. In previous editions of this book, this chapter was written explicitly for C and C++ programmers making the transition to Java. It has been rewritten for this edition to make it more generally useful, but it still contains comparisons to C and C++ for the benefit of programmers coming from those languages.[1]

> [1] Readers who want even more thorough coverage of the Java language should consider *The Java Programming Language, Second Edition*, by Ken Arnold and James Gosling (the creator of Java) (Addison Wesley Longman). And hard-core readers may want to go straight to the primary source: *The Java Language Specification*, by James Gosling, Bill Joy, and Guy Steele (Addison Wesley Longman). This specification is available in printed

book form, but is also freely available for download from Sun's web site at http://java.sun.com/docs/books/jls/. I found both documents quite helpful while writing this chapter.

This chapter documents the syntax of Java programs by starting at the very lowest level of Java syntax and building from there, covering increasingly higher orders of structure. It covers:

- The characters used to write Java programs and the encoding of those characters.

- Data types, literal values, identifiers, and other tokens that comprise a Java program.

- The operators used in Java to group individual tokens into larger expressions.

- Statements, which group expressions and other statements to form logical chunks of Java code.

- Methods (also called functions, procedures, or subroutines), which are named collections of Java statements that can be invoked by other Java code.

- Classes, which are collections of methods and fields. Classes are the central program element in Java and form the basis for object-oriented programming. Chapter 3, "Object-Oriented Programming in Java", is devoted entirely to a discussion of classes and objects.

- Packages, which are collections of related classes.

- Java programs, which consist of one or more interacting classes that may be drawn from one or more packages.

The syntax of most programming languages is complex, and Java is no exception. In general, it is not possible to document all elements of a language without referring to other elements that have not yet been discussed. For example, it is not really possible to explain in a meaningful way the operators and statements supported by Java without referring to objects. But it is also not possible to document objects thoroughly without referring to the operators and statements of the language. The process of learning Java, or any language, is therefore an iterative one. If you are new to Java (or a Java-style programming language), you may find that you benefit greatly from working through this chapter and the next *twice*, so that you can grasp the interrelated concepts.

2.1. The Unicode Character Set

Java programs are written using the Unicode character set. Unlike the 7-bit ASCII encoding, which is useful only for English, and the 8-bit ISO Latin-1 encoding, which is useful only for major Western

European languages, the 16-bit Unicode encoding can represent virtually every written language in common use on the planet. Very few text editors support Unicode, however, and in practice, most Java programs are written in plain ASCII. 16-bit Unicode characters are typically written to files using an encoding known as UTF-8, which converts the 16-bit characters into a stream of bytes. The format is designed so that plain ASCII and Latin-1 text are valid UTF-8 byte streams. Thus, you can simply write plain ASCII programs, and they will work as valid Unicode.

If you want to embed a Unicode character within a Java program that is written in plain ASCII, use the special Unicode escape sequence \u*xxxx*. That is, a backslash and a lowercase u, followed by four hexadecimal characters. For example, \u0020 is the space character, and \u3c00 is the character π. You can use Unicode characters anywhere in a Java program, including comments and variable names.

Copyright © 2001 O'Reilly & Associates. All rights reserved.

[Chapter 2: Java Syntax from
the Ground Up](#)

2.2. Comments

Java supports three types of comments. The first type is a single-line comment, which begins with the characters `//` and continues until the end of the current line. For example:

```
int i = 0;   // initialize the loop variable
```

The second kind of comment is a multiline comment. It begins with the characters `/*` and continues, over any number of lines, until the characters `*/`. Any text between the `/*` and the `*/` is ignored by the Java compiler. Although this style of comment is typically used for multiline comments, it can also be used for single-line comments. This type of comment cannot be nested (i.e., one `/* */` comment cannot appear within another one). When writing multiline comments, programmers often use extra `*` characters to make the comments stand out. Here is a typical multiline comment:

```
/*
 * Step 4: Print static methods, both public and protected,
 *         but don't list deprecated ones.
 */
```

The third type of comment is a special case of the second. If a comment begins with `/**`, it is regarded as a special *doc comment*. Like regular multiline comments, doc comments end with `*/` and cannot be nested. When you write a Java class you expect other programmers to use, use doc comments to embed documentation about the class and each of its methods directly into the source code. A program named *javadoc* extracts these comments and processes them to create online documentation for your class. A doc comment can contain HTML tags and can use additional syntax understood by *javadoc*. For example:

```
/**
 * Display a list of classes, many to a line.
 *
 * @param classes The classes to display
 * @return <tt>true</tt> on success,
 * <tt>false</tt> on failure.
 * @author David Flanagan
 */
```

See [Chapter 7, "Java Programming and Documentation Conventions"](#), for more information on the doc-comment syntax and [Chapter 8, "Java Development Tools"](#), for more information on the *javadoc* program.

Chapter 2: Java Syntax from
the Ground Up

2.3. Identifiers and Reserved Words

An *identifier* is any symbolic name that refers to something in a Java program. Class, method, parameter, and variable names are all identifiers. An identifier must begin with a letter, an underscore (_), or a Unicode currency symbol (e.g., $, £, ¥). This initial letter can be followed by any number of letters, digits, underscores, or currency symbols. Remember that Java uses the Unicode character set, which contains quite a few letters and digits other than those in the ASCII character set. The following are legal identifiers:

```
i
engine3
theCurrentTime
the_current_time
```

θ

Identifiers can include numbers, but cannot begin with a number. In addition, they cannot contain any punctuation characters other than underscores and currency characters. By convention, dollar signs and other currency characters are reserved for identifiers automatically generated by a compiler or some kind of code preprocessor. It is best to avoid these characters in your own identifiers.

Another important restriction on identifiers is that you cannot use any of the keywords and literals that are part of the Java language itself. These reserved words are listed in Table 2-1.

Table 2-1. Java Reserved Words

abstract	do	if	package	synchronized
boolean	double	implements	private	this
break	else	import	protected	throw
byte	extends	instanceof	public	throws
case	false	int	return	transient
catch	final	interface	short	true

char	finally	long	static	try
class	float	native	strictfp	void
const	for	new	super	volatile
continue	goto	null	switch	while
default				

Note that `const` and `goto` are reserved words, but aren't part of the Java language.

[Chapter 2: Java Syntax from the Ground Up](#)

2.4. Primitive Data Types

Java supports eight basic data types known as *primitive types*. In addition, it supports classes and arrays as composite data types, or reference types. Classes and arrays are documented later in this chapter. The primitive types are: a boolean type, a character type, four integer types, and two floating-point types. The four integer types and the two floating-point types differ in the number of bits that represent them, and therefore in the range of numbers they can represent. Table 2-2 summarizes these primitive data types.

Table 2-2. Java Primitive Data Types

Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	NA
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	-128 to 127
short	Signed integer	0	16 bits	-32768 to 32767
int	Signed integer	0	32 bits	-2147483648 to 2147483647
long	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

2.4.1. The boolean Type

The `boolean` type represents a truth value. There are only two possible values of this type, representing the two boolean states: on or off, yes or no, true or false. Java reserves the words `true` and `false` to represent these two boolean values.

C and C++ programmers should note that Java is quite strict about its `boolean` type: `boolean` values can never be converted to or from other data types. In particular, a `boolean` is not an integral type, and integer values cannot be used in place of a `boolean`. In other words, you cannot take shortcuts such as the following in Java:

```
if (o) {
   while(i) {
   }
}
```

Instead, Java forces you to write cleaner code by explicitly stating the comparisons you want:

```
if (o != null) {
  while(i != 0) {
  }
}
```

2.4.2. The char Type

The char type represents Unicode characters. It surprises many experienced programmers to learn that Java char values are 16 bits long, but in practice this fact is totally transparent. To include a character literal in a Java program, simply place it between single quotes (apostrophes):

```
char c = 'A';
```

You can, of course, use any Unicode character as a character literal, and you can use the \u Unicode escape sequence. In addition, Java supports a number of other escape sequences that make it easy both to represent commonly used nonprinting ASCII characters such as newline and to escape certain punctuation characters that have special meaning in Java. For example:

```
char tab = '\t', apostrophe = '\'', nul = '\000', aleph='\u05D0';
```

Table 2-3 lists the escape characters that can be used in char literals. These characters can also be used in string literals, which are covered later in this chapter.

Table 2-3. Java Escape Characters

Escape Sequence	Character Value
\b	Backspace
\t	Horizontal tab
\n	Newline
\f	Form feed
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash
\xxx	The Latin-1 character with the encoding *xxx*, where *xxx* is an octal (base 8) number between 000 and 377. The forms \x and \xx are also legal, as in '\0', but are not recommended because they can cause difficulties in string constants where the escape sequence is followed by a regular digit.

\uxxxx	The Unicode character with encoding *xxxx*, where *xxxx* is four hexadecimal digits. Unicode escapes can appear anywhere in a Java program, not only in character and string literals.

char values can be converted to and from the various integral types. Unlike byte, short, int, and long, however, char is an unsigned type. The Character class defines a number of useful static methods for working with characters, including isDigit(), isJavaLetter(), isLowerCase(), and toUpperCase().

2.4.3. Integer Types

The integer types in Java are byte, short, int, and long. As shown in Table 2-2, these four types differ only in the number of bits and, therefore, in the range of numbers each type can represent. All integral types represent signed numbers; there is no unsigned keyword as there is in C and C++.

Literals for each of these types are written exactly as you would expect: as a string of decimal digits. Although it is not technically part of the literal syntax, any integer literal can be preceded by the unary minus operator to indicate a negative number. Here are some legal integer literals:

```
0
1
123
-42000
```

Integer literals can also be expressed in hexadecimal or octal notation. A literal that begins with 0x or 0X is taken as a hexadecimal number, using the letters A to F (or a to f) as the additional digits required for base-16 numbers. Integer literals beginning with a leading 0 are taken to be octal (base-8) numbers and cannot include the digits 8 or 9. Java does not allow integer literals to be expressed in binary (base-2) notation. Legal hexadecimal and octal literals include:

```
0xff                // Decimal 255, expressed in hexadecimal
0377                // The same number, expressed in octal (base 8)
0xCAFEBABE          // A magic number used to identify Java class files
```

Integer literals are 32-bit int values unless they end with the character L or l, in which case they are 64-bit long values:

```
1234        // An int value
1234L       // A long value
0xffL       // Another long value
```

Integer arithmetic in Java is modular, which means that it never produces an overflow or an underflow when you exceed the range of a given integer type. Instead, numbers just wrap around. For example:

```
byte b1 = 127, b2 = 1;   // Largest byte is 127
byte sum = b1 + b2;      // Sum wraps to -128, which is the smallest byte
```

Neither the Java compiler nor the Java interpreter warns you in any way when this occurs. When doing integer arithmetic, you simply must ensure that the type you are using has a sufficient range for the purposes you intend. Integer

division by zero and modulo by zero are illegal and cause an `ArithmeticException` to be thrown.

Each integer type has a corresponding wrapper class: `Byte`, `Short`, `Integer`, and `Long`. Each of these classes defines `MIN_VALUE` and `MAX_VALUE` constants that describe the range of the type. The classes also define useful static methods, such as `Byte.parseByte()` and `Integer.parseInt()`, for converting strings to integer values.

2.4.4. Floating-Point Types

Real numbers in Java are represented with the `float` and `double` data types. As shown in Table 2-3, `float` is a 32-bit, single-precision floating-point value, and `double` is a 64-bit, double-precision floating-point value. Both types adhere to the IEEE 754-1985 standard, which specifies both the format of the numbers and the behavior of arithmetic for the numbers.

Floating-point values can be included literally in a Java program as an optional string of digits, followed by a decimal point and another string of digits. Here are some examples:

```
123.45
0.0
.01
```

Floating-point literals can also use exponential, or scientific, notation, in which a number is followed by the letter `e` or `E` (for exponent) and another number. This second number represents the power of ten by which the first number is multiplied. For example:

```
1.2345E02        // 1.2345 * 10^2, or 123.45
1e-6             // 1 * 10^-6, or 0.000001
6.02e23          // Avagadro's Number: 6.02 * 10^23
```

Floating-point literals are `double` values by default. To include a `float` value literally in a program, follow the number by the character `f` or `F`:

```
double d = 6.02E23;
float f = 6.02e23f;
```

Floating-point literals cannot be expressed in hexadecimal or octal notation.

Most real numbers, by their very nature, cannot be represented exactly in any finite number of bits. Thus, it is important to remember that `float` and `double` values are only approximations of the numbers they are meant to represent. A `float` is a 32-bit approximation, which results in at least 6 significant decimal digits, and a `double` is a 64-bit approximation, which results in at least 15 significant digits. In practice, these data types are suitable for most real-number computations.

In addition to representing ordinary numbers, the `float` and `double` types can also represent four special values: positive and negative infinity, zero, and NaN. The infinity values result when a floating-point computation produces a value that overflows the representable range of a `float` or `double`. When a floating-point computation underflows the representable range of a `float` or a `double`, a zero value results. The Java floating-point types make a distinction between positive zero and negative zero, depending on the direction from which the underflow occurred. In practice,

positive and negative zero behave pretty much the same. Finally, the last special floating-point value is NaN, which stands for not-a-number. The NaN value results when an illegal floating-point operation, such as 0/0, is performed. Here are examples of statements that result in these special values:

```
double inf = 1/0;          // Infinity
double neginf = -1/0;      // -Infinity
double negzero = -1/inf;   // Negative zero
double NaN = 0/0;          // NaN
```

Because the Java floating-point types can handle overflow to infinity and underflow to zero and have a special NaN value, floating-point arithmetic never throws exceptions, even when performing illegal operations, like dividing zero by zero or taking the square root of a negative number.

The `float` and `double` primitive types have corresponding classes, named `Float` and `Double`. Each of these classes defines the following useful constants: `MIN_VALUE`, `MAX_VALUE`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, and `NaN`.

The infinite floating-point values behave as you would expect. Adding or subtracting anything to or from infinity, for example, yields infinity. Negative zero behaves almost identically to positive zero, and, in fact, the = = equality operator reports that negative zero is equal to positive zero. The only way to distinguish negative zero from positive, or regular, zero is to divide by it. 1/0 yields positive infinity, but 1 divided by negative zero yields negative infinity. Finally, since NaN is not-a-number, the = = operator says that it is not equal to any other number, including itself! To check whether a `float` or `double` value is NaN, you must use the `Float.isNan()` and `Double.isNan()` methods.

2.4.5. Strings

In addition to the boolean, character, integer, and floating-point data types, Java also has a data type for working with strings of text (usually simply called *strings*). The `String` type is a class, however, and is not one of the primitive types of the language. Because strings are so commonly used, though, Java does have a syntax for including string values literally in a program. A `String` literal consists of arbitrary text within double quotes. For example:

```
"Hello, world"
"'This' is a string!"
```

String literals can contain any of the escape sequences that can appear as `char` literals (see Table 2-3). Use the `\"` sequence to include a double-quote within a `String` literal. Strings and string literals are discussed in more detail later in this chapter. Chapter 4, "The Java Platform", demonstrates some of the ways you can work with `String` objects in Java.

2.4.6. Type Conversions

Java allows conversions between integer values and floating-point values. In addition, because every character corresponds to a number in the Unicode encoding, `char` types can be converted to and from the integer and floating-point types. In fact, `boolean` is the only primitive type that cannot be converted to or from another primitive type in Java.

There are two basic types of conversions. A *widening conversion* occurs when a value of one type is converted to a wider type--one that is represented with more bits and therefore has a wider range of legal values. A *narrowing conversion* occurs when a value is converted to a type that is represented with fewer bits. Java performs widening conversions automatically when, for example, you assign an `int` literal to a `double` variable or a `char` literal to an `int` variable.

Narrowing conversions are another matter, however, and are not always safe. It is reasonable to convert the integer value 13 to a `byte`, for example, but it is not reasonable to convert 13000 to a `byte`, since `byte` can only hold numbers between -128 and 127. Because you can lose data in a narrowing conversion, the Java compiler complains when you attempt any narrowing conversion, even if the value being converted would in fact fit in the narrower range of the specified type:

```
int i = 13;
byte b = i;     // The compiler does not allow this
```

The one exception to this rule is that you can assign an integer literal (an `int` value) to a `byte` or `short` variable, if the literal falls within the range of the variable.

If you need to perform a narrowing conversion and are confident you can do so without losing data or precision, you can force Java to perform the conversion using a language construct known as a *cast*. Perform a cast by placing the name of the desired type in parentheses before the value to be converted. For example:

```
int i = 13;
byte b = (byte) i;   // Force the int to be converted to a byte
i = (int) 13.456;    // Force this double literal to the int 13
```

Casts of primitive types are most often used to convert floating-point values to integers. When you do this, the fractional part of the floating-point value is simply truncated (i.e., the floating-point value is rounded towards zero, not towards the nearest integer). The methods `Math.round()`, `Math.floor()`, and `Math.ceil()` perform other types of rounding.

The `char` type acts like an integer type in most ways, so a `char` value can be used anywhere an `int` or `long` value is required. Recall, however, that the `char` type is *unsigned*, so it behaves differently than the `short` type, even though both of them are 16 bits wide:

```
short s = (short) 0xffff; // These bits represent the number -1
char c = '\uffff';        // The same bits, representing a Unicode character
int i1 = s;               // Converting the short to an int yields -1
int i2 = c;               // Converting the char to an int yields 65535
```

is a grid that shows which primitive types can be converted to which other types and how the conversion is performed. The letter N in the table means that the conversion cannot be performed. The letter Y means that the conversion is a widening conversion and is therefore performed automatically and implicitly by Java. The letter C means that the conversion is a narrowing conversion and requires an explicit cast. Finally, the notation Y* means that the conversion is an automatic widening conversion, but that some of the least significant digits of the value may be lost by the conversion. This can happen when converting an `int` or `long` to a `float` or `double`. The floating-point types have a larger range than the integer types, so any `int` or `long` can be represented by a `float` or `double`. However,

the floating-point types are approximations of numbers and cannot always hold as many significant digits as the integer types.

Table 2-4. Java Primitive Type Conversions

Convert From:	Convert To: boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
byte	N	-	Y	C	Y	Y	Y	Y
short	N	C	-	C	Y	Y	Y	Y
char	N	C	C	-	Y	Y	Y	Y
int	N	C	C	C	-	Y	Y*	Y
long	N	C	C	C	C	-	Y*	Y*
float	N	C	C	C	C	C	-	Y
double	N	C	C	C	C	C	C	-

2.4.7. Reference Types

In addition to its eight primitive types, Java defines two additional categories of data types: classes and arrays. Java programs consist of class definitions; each class defines a new data type that can be manipulated by Java programs. For example, a program might define a class named `Point` and use it to store and manipulate X,Y points in a Cartesian coordinate system. This makes `Point` a new data type in that program. An array type represents a list of values of some other type. `char` is a data type, and an array of `char` values is another data type, written `char[]`. An array of `Point` objects is a data type, written `Point[]`. And an array of `Point` arrays is yet another type, written `Point[][]`.

As you can see, there are an infinite number of possible class and array data types. Collectively, these data types are known as *reference types*. The reason for this name will become clear later in this chapter. For now, however, what is important to understand is that class and array types differ significantly from primitive types, in that they are compound, or composite, types. A primitive data type holds exactly one value. Classes and arrays are aggregate types that contain multiple values. The `Point` type, for example, holds two `double` values representing the X and Y coordinates of the point. And `char[]` is obviously a compound type because it represents a list of characters. By their very nature, class and array types are more complicated than the primitive data types. We'll discuss classes and arrays in detail later in this chapter and examine classes in even more detail in Chapter 3, "Object-Oriented Programming in Java".

Chapter 2: Java Syntax from
the Ground Up

2.5. Expressions and Operators

So far in this chapter, we've learned about the primitive types that Java programs can manipulate and seen how to include primitive values as *literals* in a Java program. We've also used *variables* as symbolic names that represent, or hold, values. These literals and variables are the tokens out of which Java programs are built.

An *expression* is the next higher level of structure in a Java program. The Java interpreter *evaluates* an expression to compute its value. The very simplest expressions are called *primary expressions* and consist of literals and variables. So, for example, the following are all expressions:

```
1.7          // An integer literal
true         // A boolean literal
sum          // A variable
```

When the Java interpreter evaluates a literal expression, the resulting value is the literal itself. When the interpreter evaluates a variable expression, the resulting value is the value stored in the variable.

Primary expressions are not very interesting. More complex expressions are made by using *operators* to combine primary expressions. For example, the following expression uses the assignment operator to combine two primary expressions--a variable and a floating-point literal--into an assignment expression:

```
sum = 1.7
```

But operators are used not only with primary expressions; they can also be used with expressions at any level of complexity. Thus, the following are all legal expressions:

```
sum = 1 + 2 + 3*1.2 + (4 + 8)/3.0
sum/Math.sqrt(3.0 * 1.234)
(int)(sum + 33)
```

2.5.1. Operator Summary

The kinds of expressions you can write in a programming language depend entirely on the set of operators available to you. Table 2-5 summarizes the operators available in Java. The P and A columns of the table specify the precedence and associativity of each group of related operators, respectively.

Table 2-5. Java Operators

P	A	Operator	Operand Type(s)	Operation Performed
15	L	.	object, member	object member access

		[]	array, int	array element access
		(*args*)	method, arglist	method invocation
		++, − −	variable	post-increment, decrement
14	R	++, − −	variable	pre-increment, decrement
		+, −	number	unary plus, unary minus
		~	integer	bitwise complement
		!	boolean	boolean NOT
13	R	new	class, arglist	object creation
		(*type*)	type, any	cast (type conversion)
12	L	*, /, %	number, number	multiplication, division, remainder
11	L	+, −	number, number	addition, subtraction
		+	string, any	string concatenation
10	L	<<	integer, integer	left shift
		>>	integer, integer	right shift with sign extension
		>>>	integer, integer	right shift with zero extension
9	L	<, <=	number, number	less than, less than or equal
		>, >=	number, number	greater than, greater than or equal
		instanceof	reference, type	type comparison
8	L	= =	primitive, primitive	equal (have identical values)
		! =	primitive, primitive	not equal (have different values)
		= =	reference, reference	equal (refer to same object)
		! =	reference, reference	not equal (refer to different objects)
7	L	&	integer, integer	bitwise AND
		&	boolean, boolean	boolean AND

6	L	`^`	integer, integer	bitwise XOR
		`^`	boolean, boolean	boolean XOR
5	L	`|`	integer, integer	bitwise OR
		`|`	boolean, boolean	boolean OR
4	L	`&&`	boolean, boolean	conditional AND
3	L	`||`	boolean, boolean	conditional OR
2	R	`?:`	boolean, any, any	conditional (ternary) operator
1	R	`=`	variable, any	assignment
		`*=, /=, %=,`	variable, any	assignment with operation
		`+=, -=, <<=,`		
		`>>=, >>>=,`		
		`&=, ^=, |=`		

2.5.1.1. Precedence

The P column of Table 2-5 specifies the *precedence* of each operator. Precedence specifies the order in which operations are performed. Consider this expression:

```
a + b * c
```

The multiplication operator has higher precedence than the addition operator, so `a` is added to the product of `b` and `c`. Operator precedence can be thought of as a measure of how tightly operators bind to their operands. The higher the number, the more tightly they bind.

Default operator precedence can be overridden through the use of parentheses, to explicitly specify the order of operations. The previous expression can be rewritten as follows to specify that the addition should be performed before the multiplication:

```
(a + b) * c
```

The default operator precedence in Java was chosen for compatibility with C; the designers of C chose this precedence so that most expressions can be written naturally without parentheses. There are only a few common Java idioms for which parentheses are required. Examples include:

```
// Class cast combined with member access
```

```
((Integer) o).intValue();

// Assignment combined with comparison
while((line = in.readLine()) != null) { ... }

// Bitwise operators combined with comparison
if ((flags & (PUBLIC | PROTECTED)) != 0) { ... }
```

2.5.1.2. Associativity

When an expression involves several operators that have the same precedence, the operator associativity governs the order in which the operations are performed. Most operators are left-to-right associative, which means that the operations are performed from left to right. The assignment and unary operators, however, have right-to-left associativity. The A column of Table 2-5 specifies the associativity of each operator or group of operators. The value L means left to right, and R means right to left.

The additive operators are all left-to-right associative, so the expression a+b-c is evaluated from left to right: (a+b)-c. Unary operators and assignment operators are evaluated from right to left. Consider this complex expression:

```
a = b += c = -~d
```

This is evaluated as follows:

```
a = (b += (c = -(~d)))
```

As with operator precedence, operator associativity establishes a default order of evaluation for an expression. This default order can be overridden through the use of parentheses. However, the default operator associativity in Java has been chosen to yield a natural expression syntax, and you rarely need to alter it.

2.5.1.3. Operand number and type

The fourth column of Table 2-5 specifies the number and type of the operands expected by each operator. Some operators operate on only one operand; these are called unary operators. For example, the unary minus operator changes the sign of a single number:

```
-n              // The unary minus operator
```

Most operators, however, are binary operators that operate on two operand values. The – operator actually comes in both forms:

```
a - b           // The subtraction operator is a binary operator
```

Java also defines one ternary operator, often called the conditional operator. It is like an if statement inside an expression. Its three operands are separated by a question mark and a colon; the second and third operators must both be of the same type:

```
x > y ? x : y  // Ternary expression; evaluates to the larger of x and y
```

In addition to expecting a certain number of operands, each operator also expects particular types of operands. Column four of the table lists the operand types. Some of the codes used in that column require further explanation:

number

An integer, floating-point value, or character (i.e., any primitive type except `boolean`)

integer

A `byte`, `short`, `int`, `long`, or `char` value (`long` values are not allowed for the array access operator `[]`)

reference

An object or array

variable

A variable or anything else, such as an array element, to which a value can be assigned

2.5.1.4. Return type

Just as every operator expects its operands to be of specific types, each operator produces a value of a specific type. The arithmetic, increment and decrement, bitwise, and shift operators return a `double` if at least one of the operands is a `double`. Otherwise, they return a `float` if at least one of the operands is a `float`. Otherwise, they return a `long` if at least one of the operands is a `long`. Otherwise, they return an `int`, even if both operands are `byte`, `short`, or `char` types that are narrower than `int`.

The comparison, equality, and boolean operators always return `boolean` values. Each assignment operator returns whatever value it assigned, which is of a type compatible with the variable on the left side of the expression. The conditional operator returns the value of its second or third argument (which must both be of the same type).

2.5.1.5. Side effects

Every operator computes a value based on one or more operand values. Some operators, however, have *side effects* in addition to their basic evaluation. If an expression contains side effects, evaluating it changes the state of a Java program in such a way that evaluating the expression again may yield a different result. For example, the `++` increment operator has the side effect of incrementing a variable. The expression `++a` increments the variable `a` and returns the newly incremented value. If this expression is evaluated again, the value will be different. The various assignment operators also have side effects. For example, the expression `a*=2` can also be written as `a=a*2`. The value of the expression is the value of `a` multiplied by 2, but the expression also has the side effect of storing that value back into `a`. The method invocation operator `()` has side effects if the invoked method has side effects. Some methods, such as `Math.sqrt()`, simply compute and return a value without side effects of any kind. Typically, however, methods do have side effects. Finally, the `new` operator has the profound side effect of creating a new object.

2.5.1.6. Order of evaluation

When the Java interpreter evaluates an expression, it performs the various operations in an order specified by the parentheses in the expression, the precedence of the operators, and the associativity of the operators. Before any operation

is performed, however, the interpreter first evaluates the operands of the operator. (The exceptions are the &&, ||, and ?: operators, which do not always evaluate all their operands.) The interpreter always evaluates operands in order from left to right. This matters if any of the operands are expressions that contain side effects. Consider this code, for example:

```
int a = 2;
int v = ++a + ++a * ++a;
```

Although the multiplication is performed before the addition, the operands of the + operator are evaluated first. Thus, the expression evaluates to 3+4*5, or 23.

2.5.2. Arithmetic Operators

Since most programs operate primarily on numbers, the most commonly used operators are often those that perform arithmetic operations. The arithmetic operators can be used with integers, floating-point numbers, and even characters (i.e., they can be used with any primitive type other than boolean). If either of the operands is a floating-point number, floating-point arithmetic is used; otherwise, integer arithmetic is used. This matters because integer arithmetic and floating-point arithmetic differ in the way division is performed and in the way underflows and overflows are handled, for example. The arithmetic operators are:

Addition (+)

> The + operator adds two numbers. As we'll see shortly, the + operator can also be used to concatenate strings. If either operand of + is a string, the other one is converted to a string as well. Be sure to use parentheses when you want to combine addition with concatenation. For example:

```
System.out.println("Total: " + 3 + 4);    // Prints "Total: 34", not 7!
```

Subtraction (−)

> When − is used as a binary operator, it subtracts its second operand from its first. For example, 7-3 evaluates to 4. The − operator can perform unary negation.

Multiplication (*)

> The * operator multiplies its two operands. For example, 7*3 evaluates to 21.

Division (/)

> The / operator divides its first operand by its second. If both operands are integers, the result is an integer, and any remainder is lost. If either operand is a floating-point value, however, the result is a floating-point value. When dividing two integers, division by zero throws an ArithmeticException. For floating-point calculations, however, division by zero simply yields an infinite result or NaN:

```
7/3            // Evaluates to 2
7/3.0f         // Evaluates to 2.333333f
7/0            // Throws an ArithmeticException
7/0.0          // Evaluates to positive infinity
0.0/0.0        // Evaluates to NaN
```

Modulo **(%)**

The `%` operator computes the first operand modulo the second operand (i.e., it returns the remainder when the first operand is divided by the second operand an integral number of times). For example, 7%3 is 1. The sign of the result is the same as the sign of the first operand. While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, 4.3%2.1 evaluates to 0.1. When operating with integers, trying to compute a value modulo zero causes an `ArithmeticException`. When working with floating-point values, anything modulo 0.0 evaluates to `NaN`, as does infinity modulo anything.

Unary Minus **(−)**

When − is used as a unary operator, before a single operand, it performs unary negation. In other words, it converts a positive value to an equivalently negative value, and vice versa.

2.5.3. String Concatenation Operator

In addition to adding numbers, the + operator (and the related += operator) also concatenates, or joins, strings. If either of the operands to + is a string, the operator converts the other operand to a string. For example:

```
System.out.println("Quotient: " + 7/3.0f);   // Prints "Quotient: 2.3333333"
```

As a result, you must be careful to put any addition expressions in parentheses when combining them with string concatenation. If you do not, the addition operator is interpreted as a concatenation operator.

The Java interpreter has built-in string conversions for all primitive types. An object is converted to a string by invoking its `toString()` method. Some classes define custom `toString()` methods, so that objects of that class can easily be converted to strings in this way. An array is converted to a string by invoking the built-in `toString()` method, which, unfortunately, does not return a useful string representation of the array contents.

2.5.4. Increment and Decrement Operators

The ++ operator increments its single operand, which must be a variable, an element of an array, or a field of an object, by one. The behavior of this operator depends on its position relative to the operand. When used before the operand, where it is known as the *pre-increment* operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the *post-increment* operator, it increments its operand, but evaluates to the value of that operand before it was incremented.

For example, the following code sets both `i` and `j` to 2:

```
i = 1;
j = ++i;
```

But these lines set `i` to 2 and `j` to 1:

```
i = 1;
j = i++;
```

Similarly, the − − operator decrements its single numeric operand, which must be a variable, an element of an array, or a field of an object, by one. Like the ++ operator, the behavior of − − depends on its position relative to the operand. When used before the operand, it decrements the operand and returns the decremented value. When used after the operand, it decrements the operand, but returns the *un-decremented* value.

The expressions `x++` and `x− −` are equivalent to `x=x+1` and `x=x−1`, respectively, except that when using the increment and decrement operators, `x` is only evaluated once. If `x` is itself an expression with side effects, this makes a big difference. For example, these two expressions are not equivalent:

```
a[i++]++;                 // Increments an element of an array
a[i++] = a[i++] + 1;  // Adds one to an array element and stores it in another
```

These operators, in both prefix and postfix forms, are most commonly used to increment or decrement the counter that controls a loop.

2.5.5. Comparison Operators

The comparison operators consist of the equality operators that test values for equality or inequality and the relational operators used with ordered types (numbers and characters) to test for greater than and less than relationships. Both types of operators yield a `boolean` result, so they are typically used with `if` statements and `while` and `for` loops to make branching and looping decisions. For example:

```
if (o != null) ...;          // The not equals operator
while(i < a.length) ...;     // The less than operator
```

Java provides the following equality operators:

Equals (`= =`)

The `= =` operator evaluates to `true` if its two operands are equal and `false` otherwise. With primitive operands, it tests whether the operand values themselves are identical. For operands of reference types, however, it tests whether the operands refer to the same object or array. In other words, it does not test the equality of two distinct objects or arrays. In particular, note that you cannot test two distinct strings for equality with this operator.

If `= =` is used to compare two numeric or character operands that are not of the same type, the narrower operand is converted to the type of the wider operand before the comparison is done. For example, when comparing a `short` to a `float`, the `short` is first converted to a `float` before the comparison is performed. For floating-point numbers, the special negative zero value tests equal to the regular, positive zero value. Also, the special NaN (not-a-number) value is not equal to any other number, including itself. To test whether a floating-point value is NaN, use the `Float.isNan()` or `Double.isNan()` method.

Not Equals (`!=`)

The `!=` operator is exactly the opposite of the `= =` operator. It evaluates to `true` if its two primitive operands have different values or if its two reference operands refer to different objects or arrays. Otherwise, it evaluates to `false`.

The relational operators can be used with numbers and characters, but not with `boolean` values, objects, or arrays because

those types are not ordered. Java provides the following relational operators:

Less Than (`<`)

Evaluates to `true` if the first operand is less than the second.

Less Than or Equal (`<=`)

Evaluates to `true` if the first operand is less than or equal to the second.

Greater Than (`>`)

Evaluates to `true` if the first operand is greater than the second.

Greater Than or Equal (`>=`)

Evaluates to `true` if the first operand is greater than or equal to the second.

2.5.6. Boolean Operators

As we've just seen, the comparison operators compare their operands and yield a `boolean` result, which is often used in branching and looping statements. In order to make branching and looping decisions based on conditions more interesting than a single comparison, you can use the Boolean (or logical) operators to combine multiple comparison expressions into a single, more complex, expression. The Boolean operators require their operands to be `boolean` values and they evaluate to `boolean` values. The operators are:

Conditional AND (`&&`)

This operator performs a Boolean AND operation on its operands. It evaluates to `true` if and only if both its operands are `true`. If either or both operands are `false`, it evaluates to `false`. For example:

```
if (x < 10 && y > 3) ... // If both comparisons are true
```

This operator (and all the Boolean operators except the unary `!` operator) have a lower precedence than the comparison operators. Thus, it is perfectly legal to write a line of code like the one above. However, some programmers prefer to use parentheses to make the order of evaluation explicit:

```
if ((x < 10) && (y > 3)) ...
```

You should use whichever style you find easier to read.

This operator is called a conditional AND because it conditionally evaluates its second operand. If the first operand evaluates to `false`, the value of the expression is `false`, regardless of the value of the second operand. Therefore, to increase efficiency, the Java interpreter takes a shortcut and skips the second operand. Since the second operand is not guaranteed to be evaluated, you must use caution when using this operator with expressions that have side effects. On the other hand, the conditional nature of this operator allows us to write Java expressions such as the following:

```
if (data != null && i < data.length && data[i] != -1) ...
```

The second and third comparisons in this expression would cause errors if the first or second comparisons evaluated to `false`. Fortunately, we don't have to worry about this because of the conditional behavior of the `&&` operator.

Conditional OR (||)

This operator performs a Boolean OR operation on its two `boolean` operands. It evaluates to `true` if either or both of its operands are `true`. If both operands are `false`, it evaluates to `false`. Like the `&&` operator, `||` does not always evaluate its second operand. If the first operand evaluates to `true`, the value of the expression is `true`, regardless of the value of the second operand. Thus, the operator simply skips that second operand in that case.

Boolean NOT (!)

This unary operator changes the `boolean` value of its operand. If applied to a `true` value, it evaluates to `false`, and if applied to a `false` value, it evaluates to `true`. It is useful in expressions like these:

```
if (!found) ...         // found is a boolean variable declared somewhere
while (!c.isEmpty()) ... // The isEmpty() method returns a boolean value
```

Because `!` is a unary operator, it has a high precedence and often must be used with parentheses:

```
if (!(x > y && y > z))
```

Boolean AND (&)

When used with `boolean` operands, the `&` operator behaves like the `&&` operator, except that it always evaluates both operands, regardless of the value of the first operand. This operator is almost always used as a bitwise operator with integer operands, however, and many Java programmers would not even recognize its use with `boolean` operands as legal Java code.

Boolean OR (|)

This operator performs a Boolean OR operation on its two `boolean` operands. It is like the `||` operator, except that it always evaluates both operands, even if the first one is `true`. The `|` operator is almost always used as a bitwise operator on integer operands; its use with `boolean` operands is very rare.

Boolean XOR (^)

When used with `boolean` operands, this operator computes the Exclusive OR (XOR) of its operands. It evaluates to `true` if exactly one of the two operands is `true`. In other words, it evaluates to `false` if both operands are `false` or if both operands are `true`. Unlike the `&&` and `||` operators, this one must always evaluate both operands. The `^` operator is much more commonly used as a bitwise operator on integer operands. With `boolean` operands, this operator is equivalent to the `!=` operator.

2.5.7. Bitwise and Shift Operators

The bitwise and shift operators are low-level operators that manipulate the individual bits that make up an integer value. The bitwise operators are most commonly used for testing and setting individual flag bits in a value. In order to understand their behavior, you must understand binary (base-2) numbers and the twos-complement format used to represent negative

integers. You cannot use these operators with floating-point, `boolean`, array, or object operands. When used with `boolean` operands, the `&`, `|`, and `^` operators perform a different operation, as described in the previous section.

If either of the arguments to a bitwise operator is a `long`, the result is a `long`. Otherwise, the result is an `int`. If the left operand of a shift operator is a `long`, the result is a `long`; otherwise, the result is an `int`. The operators are:

Bitwise Complement (~)

The unary `~` operator is known as the bitwise complement, or bitwise NOT, operator. It inverts each bit of its single operand, converting ones to zeros and zeros to ones. For example:

```
byte b = ~12;            // ~00000110 ==> 11111001 or -13 decimal
flags = flags & ~f;      // Clear flag f in a set of flags
```

Bitwise AND (&)

This operator combines its two integer operands by performing a Boolean AND operation on their individual bits. The result has a bit set only if the corresponding bit is set in both operands. For example:

```
10 & 7                   // 00001010 & 00000111 ==> 00000010 or 2
if ((flags & f) != 0)    // Test whether flag f is set
```

When used with `boolean` operands, `&` is the infrequently used Boolean AND operator described earlier.

Bitwise OR (|)

This operator combines its two integer operands by performing a Boolean OR operation on their individual bits. The result has a bit set if the corresponding bit is set in either or both of the operands. It has a zero bit only where both corresponding operand bits are zero. For example:

```
10 | 7               // 00001010 | 00000111 ==> 00001111 or 15
flags = flags | f;   // Set flag f
```

When used with `boolean` operands, `|` is the infrequently used Boolean OR operator described earlier.

Bitwise XOR (^)

This operator combines its two integer operands by performing a Boolean XOR (Exclusive OR) operation on their individual bits. The result has a bit set if the corresponding bits in the two operands are different. If the corresponding operand bits are both ones or both zeros, the result bit is a zero. For example:

```
10 & 7                   // 00001010 ^ 00000111 ==> 00001101 or 13
```

When used with `boolean` operands, `^` is the infrequently used Boolean XOR operator.

Left Shift (<<)

The `<<` operator shifts the bits of the left operand left by the number of places specified by the right operand. High-

order bits of the left operand are lost, and zero bits are shifted in from the right. Shifting an integer left by n places is equivalent to multiplying that number by 2^n. For example:

```
10 << 1     // 00001010 << 1 = 00010100 = 20 = 10*2
7 << 3      // 00000111 << 3 = 00111000 = 56 = 7*8
-1 << 2     // 0xFFFFFFFF << 2 = 0xFFFFFFFC = -4 = -1*4
```

If the left operand is a `long`, the right operand should be between 0 and 63. Otherwise, the left operand is taken to be an `int`, and the right operand should be between 0 and 31.

Signed Right Shift (>>)

The `>>` operator shifts the bits of the left operand to the right by the number of places specified by the right operand. The low-order bits of the left operand are shifted away and are lost. The high-order bits shifted in are the same as the original high-order bit of the left operand. In other words, if the left operand is positive, zeros are shifted into the high-order bits. If the left operand is negative, ones are shifted in instead. This technique is known as *sign extension*; it is used to preserve the sign of the left operand. For example:

```
10 >> 1      // 00001010 >> 1 = 00000101 = 5 = 10/2
27 >> 3      // 00011011 >> 3 = 00000011 = 3 = 27/8
-50 >> 2     // 11001110 >> 2 = 11110011 = -13 != -50/4
```

If the left operand is positive and the right operand is n, the `>>` operator is the same as integer division by 2^n.

Unsigned Right Shift (>>>)

This operator is like the `>>` operator, except that it always shifts zeros into the high-order bits of the result, regardless of the sign of the left-hand operand. This technique is called *zero extension*•; it is appropriate when the left operand is being treated as an unsigned value (despite the fact that Java integer types are all signed). Examples:

```
-50 >>> 2    // 11001110 >>> 2 = 00110011 = 51
0xff >>> 4   // 11111111 >>> 4 = 00001111 = 15  = 255/16
```

2.5.8. Assignment Operators

The assignment operators store, or assign, a value into some kind of variable. The left operand must evaluate to an appropriate local variable, array element, or object field. The right side can be any value of a type compatible with the variable. An assignment expression evaluates to the value that is assigned to the variable. More importantly, however, the expression has the side effect of actually performing the assignment. Unlike all other binary operators, the assignment operators are right-associative, which means that the assignments in `a=b=c` are performed right-to-left, as follows: `a=(b=c)`.

The basic assignment operator is `=`. Do not confuse it with the equality operator, `= =`. In order to keep these two operators distinct, I recommend that you read `=` as "is assigned the value."

In addition to this simple assignment operator, Java also defines 11 other operators that combine assignment with the 5 arithmetic operators and the 6 bitwise and shift operators. For example, the `+=` operator reads the value of the left variable, adds the value of the right operand to it, stores the sum back into the left variable as a side effect, and returns the sum as the

value of the expression. Thus, the expression `x+=2` is almost the same `x=x+2`. The difference between these two expressions is that when you use the `+=` operator, the left operand is evaluated only once. This makes a difference when that operand has a side effect. Consider the following two expressions, which are not equivalent:

```
a[i++] += 2;
a[i++] = a[i++] + 2;
```

The general form of these combination assignment operators is:

```
var op= value
```

This is equivalent (unless there are side effects in `var`) to:

```
var = var op value
```

The available operators are:

```
+=    -=    *=    /=    %=      // Arithmetic operators plus assignment
&=    |=    ^=                  // Bitwise operators plus assignment
<<=   >>=   >>>=                // Shift operators plus assignment
```

The most commonly used operators are `+=` and `- =`, although `&=` and `|=` can also be useful when working with `boolean` flags. For example:

```
i += 2;             // Increment a loop counter by 2
c -= 5;             // Decrement a counter by 5
flags |= f;         // Set a flag f in an integer set of flags
flags &= ~f;        // Clear a flag f in an integer set of flags
```

2.5.9. The Conditional Operator

The conditional operator `? :` is a somewhat obscure ternary (three-operand) operator inherited from C. It allows you to embed a conditional within an expression. You can think of it as the operator version of the `if/else` statement. The first and second operands of the conditional operator are separated by a question mark (`?`), while the second and third operands are separated by a colon (`:`). The first operand must evaluate to a `boolean` value. The second and third operands can be of any type, but they must both be of the same type.

The conditional operator starts by evaluating its first operand. If it is `true`, the operator evaluates its second operand and uses that as the value of the expression. On the other hand, if the first operand is `false`, the conditional operator evaluates and returns its third operand. The conditional operator never evaluates both its second and third operand, so be careful when using expressions with side effects with this operator. Examples of this operator are:

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

Note that the `? :` operator has lower precedence than all other operators except the assignment operators, so parentheses are not usually necessary around the operands of this operator. Many programmers find conditional expressions easier to read if the first operand is placed within parentheses, however. This is especially true because the conditional `if` statement always

has its conditional expression written within parentheses.

2.5.10. The instanceof Operator

The `instanceof` operator requires an object or array value as its left operand and the name of a reference type as its right operand. It evaluates to `true` if the object or array is an *instance* of the specified type; it returns `false` otherwise. If the left operand is `null`, `instanceof` always evaluates to `false`. If an `instanceof` expression evaluates to `true`, it means that you can safely cast and assign the left operand to a variable of the type of the right operand.

The `instanceof` operator can be used only with array and object types and values, not primitive types and values. Object and array types are discussed in detail later in this chapter. Examples of `instanceof` are:

```
"string" instanceof String      // True: all strings are instances of String
"" instanceof Object            // True: strings are also instances of Object
new int[] {1} instanceof int[]  // True: the array value is an int array
new int[] {1} instanceof byte[] // False: the array value is not a byte array
new int[] {1} instanceof Object // True: all arrays are instances of Object
null instanceof String          // False: null is never instanceof anything

// Use instanceof to make sure that it is safe to cast an object
if (object instanceof Point) {
  Point p = (Point) object;
}
```

2.5.11. Special Operators

There are five language constructs in Java that are sometimes considered operators and sometimes considered simply part of the basic language syntax. These "operators" are listed in Table 2-5 in order to show their precedence relative to the other true operators. The use of these language constructs is detailed elsewhere in this chapter, but is described briefly here, so that you can recognize these constructs when you encounter them in code examples:

Object member access (`.`)

An *object* is a collection of data and methods that operate on that data; the data fields and methods of an object are called its members. The dot (`.`) operator accesses these members. If `o` is an expression that evaluates to an object reference, and `f` is the name of a field of the object, `o.f` evaluates to the value contained in that field. If `m` is the name of a method, `o.m` refers to that method and allows it to be invoked using the `()` operator shown later.

Array element access (`[]`)

An *array* is a numbered list of values. Each element of an array can be referred to by its number, or *index*. The `[]` operator allows you to refer to the individual elements of an array. If `a` is an array, and `i` is an expression that evaluates to an int, `a[i]` refers to one of the elements of `a`. Unlike other operators that work with integer values, this operator restricts array index values to be of type `int` or narrower.

Method invocation (`()`)

A *method* is a named collection of Java code that can be run, or *invoked*, by following the name of the method with zero or more comma-separated expressions contained within parentheses. The values of these expressions are the

arguments to the method. The method processes the arguments and optionally returns a value that becomes the value of the method invocation expression. If `o.m` is a method that expects no arguments, the method can be invoked with `o.m()`. If the method expects three arguments, for example, it can be invoked with an expression such as `o.m(x,y,z)`. Before the Java interpreter invokes a method, it evaluates each of the arguments to be passed to the method. These expressions are guaranteed to be evaluated in order from left to right (which matters if any of the arguments have side effects).

Object creation (`new`)

In Java, objects are created with the `new` operator, which is followed by the type of the object to be created and a parenthesized list of arguments to be passed to the object *constructor*. A constructor is a special method that initializes a newly created object, so the object creation syntax is similar to the Java method invocation syntax. For example:

```
new ArrayList();
new Point(1,2)
```

Type conversion or casting (`()`)

As we've already seen, parentheses can also be used as an operator to perform narrowing type conversions, or casts. The first operand of this operator is the type to be converted to; it is placed between the parentheses. The second operand is the value to be converted; it follows the parentheses. For example:

```
(byte) 28        // An integer literal cast to a byte type
(int) (x + 3.14f)  // A floating-point sum value cast to an integer value
(String)h.get(k)   // A generic object cast to a more specific string type
```

[Chapter 2: Java Syntax from the Ground Up](#)

2.6. Statements

A *statement* is a single "command" that is executed by the Java interpreter. By default, the Java interpreter runs one statement after another, in the order they are written. Many of the statements defined by Java, however, are flow-control statements, such as conditionals and loops, that alter this default order of execution in well-defined ways. [Table 2-6](#) summarizes the statements defined by Java.

Table 2-6. Java Statements

Statement	Purpose	Syntax
expression	side effects	*var=expr*; *expr++*; *method*(); new*Type*();
compound	group statements	{ *statements* }
empty	do nothing	;
labeled	name a statement	*label* : *statement*
variable	declare a variable	[final] *type name* [= *value*] [, *name* [= *value*]] ...;
if	conditional	if (*expr*) *statement* [else *statement*]
switch	conditional	switch (*expr*) { [case*expr*:*statements*] ... [default:*statements*] }
while	loop	while (*expr*) *statement*
do	loop	do *statement* while (*expr*);
for	simplified loop	for (*init* ; *test* ; *increment*) *statement*
break	exit block	break [*label*];
continue	restart loop	continue [*label*];
return	end method	return [*expr*];
synchronized	critical section	synchronized (*expr*) { *statements* }
throw	throw exception	throw *expr* ;
try	handle exception	try {*statements*} [catch (*typename*) {*statements*}] ... [finally {*statements*}]

2.6.1. Expression Statements

As we saw earlier in the chapter, certain types of Java expressions have side effects. In other words, they do not simply evaluate to some value, but also change the program state in some way. Any expression with side effects can be used as a statement simply by following it with a semicolon. The legal types of expression statements are assignments, increments and

decrements, method calls, and object creation. For example:

```
a = 1;                              // Assignment
x *= 2;                             // Assignment with operation
i++;                                // Post-increment
--c;                                // Pre-decrement
System.out.println("statement");    // Method invocation
```

2.6.2. Compound Statements

A *compound statement* is any number and kind of statements grouped together within curly braces. You can use a compound statement anywhere a *statement* is required by Java syntax:

```
for(int i = 0; i < 10; i++) {
  a[i]++;                       // Body of this loop is a compound
  statement. b[i]--;            // It consists of two expression statements
}                               // within curly braces.
```

2.6.3. The Empty Statement

An *empty statement* in Java is written as a single semicolon. The empty statement doesn't do anything, but the syntax is occasionally useful. For example, you can use it to indicate an empty loop body of a `for` loop:

```
for(int i = 0; i < 10; a[i++]++)  // Increment array elements
    /* empty */;                  // Loop body is empty statement
```

2.6.4. Labeled Statements

A *labeled statement* is simply a statement that has been given a name by prepending a identifier and a colon to it. Labels are used by the `break` and `continue` statements. For example:

```
rowLoop: for(int r = 0; r < rows.length; r++) {       // A labeled loop
  colLoop: for(int c = 0; c < columns.length; c++) {  // Another one
    break rowLoop;                                     // Use a label
  }
}
```

2.6.5. Local Variable Declaration Statements

A *local variable*, often simply called a variable, is a symbolic name for a location where a value can be stored that is defined within a method or compound statement. All variables must be declared before they can be used; this is done with a variable declaration statement. Because Java is a strongly typed language, a variable declaration specifies the type of the variable, and only values of that type can be stored in the variable.

In its simplest form, a variable declaration specifies a variable's type and name:

```
int counter;
String s;
```

A variable declaration can also include an *initializer* : an expression that specifies an initial value for the variable. For example:

```
int i = 0;
String s = readLine();
int[] data = {x+1, x+2, x+3};  // Array initializers are documented later
```

The Java compiler does not allow you to use a variable that has not been initialized, so it is usually convenient to combine variable declaration and initialization into a single statement. The initializer expression need not be a literal value or a constant expression that can be evaluated by the compiler; it can be an arbitrarily complex expression whose value is computed when the program is run.

A single variable declaration statement can declare and initialize more than one variable, but all variables must be of the same type. Variable names and optional initializers are separated from each other with commas:

```
int i, j, k;
float x = 1.0, y = 1.0;
String question = "Really Quit?", response;
```

In Java 1.1 and later, variable declaration statements can begin with the `final` keyword. This modifier specifies that once an initial value is specified for the variable, that value is never allowed to change:

```
final String greeting = getLocalLanguageGreeting();
```

C programmers should note that Java variable declaration statements can appear anywhere in Java code; they are not restricted to the beginning of a method or block of code. Local variable declarations can also be integrated with the *initialize* portion of a `for` loop, as we'll discuss shortly.

Local variables can be used only within the method or block of code in which they are defined. This is called their *scope* or *lexical scope* :

```
void method() {                 // A generic method
  int i = 0;                    // Declare variable i
  while (i < 10) {              // i is in scope here
    int j = 0;                  // Declare j; i and j are in scope here
  }                             // j is no longer in scope; can't use it anymore
  System.out.println(i);        // i is still in scope here
}                               // The scope of i ends here
```

2.6.6. The if/else Statement

The `if` statement is the fundamental control statement that allows Java to make decisions or, more precisely, to execute statements conditionally. The `if` statement has an associated expression and statement. If the expression evaluates to `true`, the interpreter executes the statement. If the expression evaluates to `false`, however, the interpreter skips the statement. For example:

```
if (username == null)           // If username is null,
  username = "John Doe";        // define it.
```

Although they look extraneous, the parentheses around the expression are a required part of the syntax for the `if` statement.

As I already mentioned, a block of statements enclosed in curly braces is itself a statement, so we can also write `if` statements that look as follows:

```
if ((address == null) || (address.equals(""))) {
  address = "[undefined]";
  System.out.println("WARNING: no address specified.");
}
```

An `if` statement can include an optional `else` keyword that is followed by a second statement. In this form of the statement, the expression is evaluated, and, if it is `true`, the first statement is executed. Otherwise, the second statement is executed. For example:

```
if (username != null)
  System.out.println("Hello " + username);
else {
  username = askQuestion("What is your name?");
  System.out.println("Hello " + username + ". Welcome!");
}
```

When you use nested `if`/`else` statements, some caution is required to ensure that the `else` clause goes with the appropriate `if` statement. Consider the following lines:

```
if (i == j)
  if (j == k)
    System.out.println("i equals k");
else
  System.out.println("i doesn't equal j");    // WRONG!!
```

In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear (except from the hint given by the indentation) which `if` the `else` goes with. And in this example, the indentation hint is wrong. The rule is that an `else` clause like this is associated with the nearest `if` statement. Properly indented, this code looks like this:

```
if (i == j)
  if (j == k)
    System.out.println("i equals k");
  else
    System.out.println("i doesn't equal j");    // WRONG!!
```

This is legal code, but it is clearly not what the programmer had in mind. When working with nested `if` statements, you should use curly braces to make your code easier to read. Here is a better way to write the code:

```
if (i == j) {
  if (j == k)
    System.out.println("i equals k");
}
else {
  System.out.println("i doesn't equal j");
```

```
}
```

2.6.6.1. The else if clause

The `if/else` statement is useful for testing a condition and choosing between two statements or blocks of code to execute. But what about when you need to choose between several blocks of code? This is typically done with an `else if` clause, which is not really new syntax, but a common idiomatic usage of the standard `if/else` statement. It looks like this:

```
if (n == 1) {
    // Execute code block #1
}
else if (n == 2) {
    // Execute code block #2
}
else if (n == 3) {
    // Execute code block #3
}
else {
    // If all else fails, execute block #4
}
```

There is nothing special about this code. It is just a series of `if` statements, where each `if` is part of the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing these statements out in their fully nested form:

```
if (n == 1) {
  // Execute code block #1
}
else {
  if (n == 2) {
    // Execute code block #2
  }
  else {
    if (n == 3) {
      // Execute code block #3
    }
    else {
      // If all else fails, execute block #4
    }
  }
}
```

2.6.7. The switch Statement

An `if` statement causes a branch in the flow of a program's execution. You can use multiple `if` statements, as shown in the previous section, to perform a multiway branch. This is not always the best solution, however, especially when all of the branches depend on the value of a single variable. In this case, it is inefficient to repeatedly check the value of the same variable in multiple `if` statements.

A better solution is to use a `switch` statement, which is inherited from the C programming language. Although the syntax

of this statement is not nearly as elegant as other parts of Java, the brute practicality of the construct makes it worthwhile. If you are not familiar with the `switch` statement itself, you may at least be familiar with the basic concept, under the name computed goto or jump table. A `switch` statement has an integer expression and a body that contains various numbered entry points. The expression is evaluated, and control jumps to the entry point specified by that value. For example, the following `switch` statement is equivalent to the repeated `if` and `else`/`if` statements shown in the previous section:

```
switch(n) {
  case 1:                             // Start here if n == 1
    // Execute code block #1
    break;                            // Stop here
  case 2:                             // Start here if n == 2
    // Execute code block #2
    break;                            // Stop here
  case 3:                             // Start here if n == 3
    // Execute code block #3
    break;                            // Stop here
  default:                            // If all else fails...
                                      // Execute code block #4
    break;                            // Stop here
}
```

As you can see from the example, the various entry points into a `switch` statement are labeled either with the keyword `case`, followed by an integer value and a colon, or with the special `default` keyword, followed by a colon. When a `switch` statement executes, the interpreter computes the value of the expression in parentheses and then looks for a `case` label that matches that value. If it finds one, the interpreter starts executing the block of code at the first statement following the `case` label. If it does not find a `case` label with a matching value, the interpreter starts execution at the first statement following a special-case `default:` label. Or, if there is no `default:` label, the interpreter skips the body of the `switch` statement altogether.

Note the use of the `break` keyword at the end of each `case` in the previous code. The `break` statement is described later in this chapter, but, in this case, it causes the interpreter to exit the body of the `switch` statement. The `case` clauses in a `switch` statement specify only the *starting point* of the desired code. The individual cases are not independent blocks of code, and they do not have any implicit ending point. Therefore, you must explicitly specify the end of each case with a `break` or related statement. In the absence of `break` statements, a `switch` statement begins executing code at the first statement after the matching `case` label and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that falls through from one `case` label to the next, but 99% of the time you should be careful to end every `case` and `default` section with a statement that causes the `switch` statement to stop executing. Normally you use a `break` statement, but `return` and `throw` also work.

A `switch` statement can have more than one `case` clause labeling the same statement. Consider the `switch` statement in the following method:

```
boolean parseYesOrNoResponse(char response) {
  switch(response) {
    case 'y':
    case 'Y': return true;
    case 'n':
    case 'N': return false;
    default: throw new IllegalArgumentException("Response must be Y or N");
  }
}
```

There are some important restrictions on the `switch` statement and its `case` labels. First, the expression associated with a `switch` statement must have a `byte`, `char`, `short`, or `int` value. The floating-point and `boolean` types are not supported, and neither is `long`, even though `long` is an integer type. Second, the value associated with each `case` label must be a constant value or a constant expression the compiler can evaluate. A `case` label cannot contain a runtime expressions involving variables or method calls, for example. Third, the `case` label values must be within the range of the data type used for the `switch` expression. And finally, it is obviously not legal to have two or more `case` labels with the same value or more than one `default` label.

2.6.8. The while Statement

Just as the `if` statement is the basic control statement that allows Java to make decisions, the `while` statement is the basic statement that allows Java to perform repetitive actions. It has the following syntax:

```
while (expression)
   statement
```

The `while` statement works by first evaluating the *expression*. If it is `false`, the interpreter skips the *statement* associated with the loop and moves to the next statement in the program. If it is `true`, however, the *statement* that forms the body of the loop is executed, and the *expression* is reevaluated. Again, if the value of *expression* is `false`, the interpreter moves on to the next statement in the program; otherwise it executes the *statement* again. This cycle continues while the *expression* remains `true` (i.e., until it evaluates to `false`), at which point the `while` statement ends, and the interpreter moves on to the next statement. You can create an infinite loop with the syntax `while(true)`.

Here is an example `while` loop that prints the numbers 0 to 9:

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

As you can see, the variable `count` starts off at 0 in this example and is incremented each time the body of the loop runs. Once the loop has executed 10 times, the expression becomes `false` (i.e., `count` is no longer less than 10), the `while` statement finishes, and the Java interpreter can move to the next statement in the program. Most loops have a counter variable like `count`. The variable names `i`, `j`, and `k` are commonly used as a loop counters, although you should use more descriptive names if it makes your code easier to understand.

2.6.9. The do Statement

A `do` loop is much like a `while` loop, except that the loop expression is tested at the bottom of the loop, rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

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

There are a couple of differences to notice between the `do` loop and the more ordinary `while` loop. First, the `do` loop requires both the `do` keyword to mark the beginning of the loop and the `while` keyword to mark the end and introduce the

loop condition. Also, unlike the `while` loop, the `do` loop is terminated with a semicolon. This is because the `do` loop ends with the loop condition, rather than simply ending with a curly brace that marks the end of the loop body. The following `do` loop prints the same output as the `while` loop shown above:

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

Note that the `do` loop is much less commonly used than its `while` cousin. This is because, in practice, it is unusual to encounter a situation where you are sure you always want a loop to execute at least once.

2.6.10. The for Statement

The `for` statement provides a looping construct that is often more convenient than the `while` and `do` loops. The `for` statement takes advantage of a common looping pattern. Most loops have a counter, or state variable of some kind, that is initialized before the loop starts, tested to determine whether to execute the loop body, and then incremented, or updated somehow, at the end of the loop body before the test expression is evaluated again. The initialization, test, and update steps are the three crucial manipulations of a loop variable, and the `for` statement makes these three steps an explicit part of the loop syntax:

```
for(initialize ; test ; increment)
    statement
```

This `for` loop is basically equivalent to the following `while` loop:[2]

> [2] As you'll see when we consider the `continue` statement, this `while` loop is not exactly equivalent to the `for` loop. We'll discuss how to write the true equivalent when we talk about the `try/catch/finally` statement.

```
initialize;
while(test) {
   statement;
   increment;
}
```

Placing the *initialize*, *test*, and *increment* expressions at the top of a `for` loop makes it especially easy to understand what the loop is doing, and it prevents mistakes such as forgetting to initialize or increment the loop variable. The interpreter discards the values of the *initialize* and *increment* expressions, so in order to be useful, these expressions must have side effects. *initialize* is typically an assignment expression, while *increment* is usually an increment, decrement, or some other assignment.

The following `for` loop prints the numbers 0 to 9, just as the previous `while` and `do` loops have done:

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

Notice how this syntax places all the important information about the loop variable on a single line, making it very clear how the loop executes. Placing the increment expression in the `for` statement itself also simplifies the body of the loop to a single statement; we don't even need to use curly braces to produce a statement block.

The `for` loop supports some additional syntax that makes it even more convenient to use. Because many loops use their loop variables only within the loop, the `for` loop allows the *initialize* expression to be a full variable declaration, so that the variable is scoped to the body of the loop and is not visible outside of it. For example:

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

Furthermore, the `for` loop syntax does not restrict you to writing loops that use only a single variable. Both the *initialize* and *increment* expressions of a `for` loop can use a comma to separate multiple initializations and increment expressions. For example:

```
for(int i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

Even though all the examples so far have counted numbers, `for` loops are not restricted to loops that count numbers. For example, you might use a `for` loop to iterate through the elements of a linked list:

```
for(Node n = listHead; n != null; n = n.nextNode())
   process(n);
```

The *initialize*, *test*, and *increment* expressions of a `for` loop are all optional; only the semicolons that separate the expressions are required. If the *test* expression is omitted, it is assumed to be `true`. Thus, you can write an infinite loop as `for(;;)`.

2.6.11. The break Statement

A `break` statement causes the Java interpreter to skip immediately to the end of a containing statement. We have already seen the `break` statement used with the `switch` statement. The `break` statement is most often written as simply the keyword `break` followed by a semicolon:

```
break;
```

When used in this form, it causes the Java interpreter to immediately exit the innermost containing `while`, `do`, `for`, or `switch` statement. For example:

```
for(int i = 0; i < data.length; i++) {  // Loop through the data array.
  if (data[i] == target) {               // When we find what we're looking for,
    index = i;                           // remember where we found it
    break;                               // and stop looking!
  }
}   // The Java interpreter goes here after executing break
```

The `break` statement can also be followed by the name of a containing labeled statement. When used in this form, `break` causes the Java interpreter to immediately exit from the named block, which can be any kind of statement, not just a loop or `switch`. For example:

```
testfornull: if (data != null) {                  // If the array is defined,
   for(int row = 0; row < numrows; row++) {    // loop through one dimension,
      for(int col = 0; col < numcols; col++) { // then loop through the other.
         if (data[row][col] == null)           // If the array is missing data,
            break testfornull;                 // treat the array as undefined.
      }
   }
}  // Java interpreter goes here after executing break testfornull
```

2.6.12. The continue Statement

While a `break` statement exits a loop, a `continue` statement quits the current iteration of a loop and starts the next one. `continue`, in both its unlabeled and labeled forms, can be used only within a `while`, `do`, or `for` loop. When used without a label, `continue` causes the innermost loop to start a new iteration. When used with a label that is the name of a containing loop, it causes the named loop to start a new iteration. For example:

```
for(int i = 0; i < data.length; i++) {  // Loop through data.
   if (data[i] == -1)                    // If a data value is missing,
      continue;                          // skip to the next iteration.
   process(data[i]);                     // Process the data value.
}
```

`while`, `do`, and `for` loops differ slightly in the way that `continue` starts a new iteration:

- With a `while` loop, the Java interpreter simply returns to the top of the loop, tests the loop condition again, and, if it evaluates to `true`, executes the body of the loop again.

- With a `do` loop, the interpreter jumps to the bottom of the loop, where it tests the loop condition to decide whether to perform another iteration of the loop.

- With a `for` loop, the interpreter jumps to the top of the loop, where it first evaluates the *increment* expression and then evaluates the *test* expression to decide whether to loop again. As you can see, the behavior of a `for` loop with a `continue` statement is different from the behavior of the "basically equivalent" `while` loop I presented earlier; *increment* gets evaluated in the `for` loop, but not in the equivalent `while` loop.

2.6.13. The return Statement

A `return` statement tells the Java interpreter to stop executing the current method. If the method is declared to return a value, the `return` statement is followed by an expression. The value of the expression becomes the return value of the method. For example, the following method computes and returns the square of a number:

```
double square(double x) {       // A method to compute x squared
   return x * x;                 // Compute and return a value
}
```

Some methods are declared `void` to indicate they do not return any value. The Java interpreter runs methods like this by executing its statements Nutents one by one until it reaches the end of the method. After executing the last statement, the interpreter returns implicitly. Sometimes, however, a `void` method has to return explicitly before reaching the last statement. In this case, it can use the `return` statement by itself, without any expression. For example, the following method prints, but does

not return, the square root of its argument. If the argument is a negative number, it returns without printing anything:

```
void printSquareRoot(double x) {      // A method to print square root of x
  if (x < 0) return;                  // If x is negative, return explicitly
  System.out.println(Math.sqrt(x));   // Print the square root of x
}                                     // End of method: return implicitly
```

2.6.14. The synchronized Statement

Since Java is a multithreaded system, you must often take care to prevent multiple threads from modifying an object simultaneously in a way that might corrupt the object's state. Sections of code that must not be executed simultaneously are known as *critical sections*. Java provides the `synchronized` statement to protect these critical sections. The syntax is:

```
synchronized ( expression ) {
  statements
}
```

expression is an expression that must evaluate to an object or an array. The *statements* constitute the code of the critical section and must be enclosed in curly braces. Before executing the critical section, the Java interpreter first obtains an exclusive lock on the object or array specified by *expression*. It holds the lock until it is finished running the critical section, then releases it. While a thread holds the lock on an object, no other thread can obtain that lock. Therefore, no other thread can execute this or any other critical sections that require a lock on the same object. If a thread cannot immediately obtain the lock required to execute a critical section, it simply waits until the lock becomes available.

Note that you do not have to use the `synchronized` statement unless your program creates multiple threads that share data. If only one thread ever accesses a data structure, there is no need to protect it with `synchronized`. When you do have to use `synchronized`, it might be in code like the following:

```
public static void SortIntArray(int[] a) {
  // Sort the array a. This is synchronized so that some other thread
  // cannot change elements of the array while we're sorting it (at
  // least not other threads that protect their changes to the array
  // with synchronized).
  synchronized (a) {
    // Do the array sort here...
  }
}
```

The `synchronized` keyword is also available as a modifier in Java and is more commonly used in this form than as a statement. When applied to a method, the `synchronized` keyword indicates that the entire method is a critical section. For a `synchronized` class method (a static method), Java obtains an exclusive lock on the class before executing the method. For a `synchronized` instance method, Java obtains an exclusive lock on the class instance. (Class and instance methods are discussed in Chapter 3, "Object-Oriented Programming in Java".)

2.6.15. The throw Statement

An *exception* is a signal that indicates some sort of exceptional condition or error has occurred. To *throw* an exception is to signal an exceptional condition. To *catch* an exception is to handle it--to take whatever actions are necessary to recover from it.

In Java, the `throw` statement is used to throw an exception:

```
throw expression ;
```

The *expression* must evaluate to an exception object that describes the exception or error that has occurred. We'll talk more about types of exceptions shortly; for now, all you need to know is that an exception is represented by an object. Here is some example code that throws an exception:

```
public static double factorial(int x) {
  if (x < 0)
    throw new IllegalArgumentException("x must be >= 0"));
  double fact;
  for(fact=1.0; x > 1; fact *= x, x--)
    /* empty */ ;            // Note use of the empty statement
  return fact;
}
```

When the Java interpreter executes a `throw` statement, it immediately stops normal program execution and starts looking for an exception handler that can catch, or handle, the exception. Exception handlers are written with the `try/catch/finally` statement, which is described in the next section. The Java interpreter first looks at the enclosing block of code to see if it has an associated exception handler. If so, it exits that block of code and starts running the exception-handling code associated with the block. After running the exception handler, the interpreter continues execution at the statement immediately following the handler code.

If the enclosing block of code does not have an appropriate exception handler, the interpreter checks the next higher enclosing block of code in the method. This continues until a handler is found. If the method does not contain an exception handler that can handle the exception thrown by the `throw` statement, the interpreter stops running the current method and returns to the caller. Now the interpreter starts looking for an exception handler in the blocks of code of the calling method. In this way, exceptions propagate up through the lexical structure of Java methods, up the call stack of the Java interpreter. If the exception is never caught, it propagates all the way up to the `main()` method of the program. If it is not handled in that method, the Java interpreter prints an error message, prints a stack trace to indicate where the exception occurred, and then exits.

2.6.15.1. Exception types

An exception in Java is an object. The type of this object is `java.langThrowable`, or more commonly, some subclass of `Throwable` that more specifically describes the type of exception that occurred.[3]`Throwable` has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. Exceptions that are subclasses of `Error` generally indicate unrecoverable problems: the virtual machine has run out of memory, or a class file is corrupted and cannot be read, for example. Exceptions of this sort can be caught and handled, but it is rare to do so. Exceptions that are subclasses of `Exception`, on the other hand, indicate less severe conditions. These are exceptions that can be reasonably caught and handled. They include such exceptions as `java.io.EOFException`, which signals the end of a file, and `java.lang.ArrayIndexOutOfBoundsException`, which indicates that a program has tried to read past the end of an array. In this book, I use the term "exception" to refer to any exception object, regardless of whether the type of that exception is `Exception` or `Error`.

[3]We haven't talked about subclasses yet; they are covered in detail in Chapter 3, "Object-Oriented Programming in Java".

Since an exception is an object, it can contain data, and its class can define methods that operate on that data. The `Throwable` class and all its subclasses include a `String` field that stores a human-readable error message that describes the exceptional condition. It's set when the exception object is created and can be read from the exception with the `getMessage()` method. Most exceptions contain only this single message, but a few add other data. The `java.io.InterruptedIOException`, for example, adds a field named `bytesTransferred` that specifies how much input or output was completed before the exceptional condition interrupted it.

2.6.15.2. Declaring exceptions

In addition to making a distinction between `Error` and `Exception` classes, the Java exception-handling scheme also makes a distinction between checked and unchecked exceptions. Any exception object that is an `Error` is unchecked. Any exception object that is an `Exception` is checked, unless it is a subclass of `java.lang.RuntimeException`, in which case it is unchecked. (`RuntimeException` is a subclass of `Exception`.) The reason for this distinction is that virtually any method can throw an unchecked exception, at essentially any time. There is no way to predict an `OutOfMemoryError`, for example, and any method that uses objects or arrays can throw a `NullPointerException` if it is passed an invalid `null` argument. Checked exceptions, on the other hand, arise only in specific, well-defined circumstances. If you try to read data from a file, for example, you must at least consider the possibility that a `FileNotFoundException` will be thrown if the specified file cannot be found.

Java has different rules for working with checked and unchecked exceptions. If you write a method that throws a checked exception, you must use a `throws` clause to declare the exception in the method signature. The reason these types of exceptions are called checked exceptions is that the Java compiler checks to make sure you have declared them in method signatures and produces a compilation error if you have not. The `factorial()` method shown earlier throws an exception of type `java.lang.IllegalArgumentException`. This is a subclass of `RuntimeException`, so it is an unchecked exception, and we do not have to declare it with a `throws` clause (although we can if we want to be explicit).

Even if you never `throw` an exception yourself, there are times when you must use a `throws` clause to declare an exception. If your method calls a method that can throw a checked exception, you must either include exception-handling code to handle that exception or use `throws` to declare that your method can also throw that exception.

How do you know if the method you are calling can throw a checked exception? You can look at its method signature to find out. Or, failing that, the Java compiler will tell you (by reporting a compilation error) if you've called a method whose exceptions you must handle or declare. The following method reads the first line of text from a named file. It uses methods that can throw various types of `java.io.IOException` objects, so it declares this fact with a `throws` clause:

```
public static String readFirstLine(String filename) throws IOException {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  return in.readLine();
}
```

We'll talk more about method declarations and method signatures later in this chapter.

2.6.16. The try/catch/finally Statement

The `try/catch/finally` statement is Java's exception-handling mechanism. The `try` clause of this statement establishes a block of code for exception handling. This `try` block is followed by zero or more `catch` clauses, each of which is a block of statements designed to handle a specific type of exception. The `catch` clauses are followed by an optional `finally` block that contains cleanup code guaranteed to be executed regardless of what happens in the `try` block. Both the `catch` and `finally` clauses are optional, but every `try` block must be accompanied by at least one or the other.

The `try`, `catch`, and `finally` blocks all begin and end with curly braces. These are a required part of the syntax and cannot be omitted, even if the clause contains only a single statement.

The following code illustrates the syntax and purpose of the `try/catch/finally` statement:

```
try {
  // Normally this code runs from the top of the block to the bottom
  // without problems. But it can sometimes throw an exception,
  // either directly with a throw statement or indirectly by calling
  // a method that throws an exception.
}
catch (SomeException e1) {
  // This block contains statements that handle an exception object
  // of type SomeException or a subclass of that type. Statements in
  // this block can refer to that exception object by the name e1.
}
catch (AnotherException e2) {
  // This block contains statements that handle an exception object
  // of type AnotherException or a subclass of that type. Statements
  // in this block can refer to that exception object by the name e2.
}


finally {
  // This block contains statements that are always executed
  // after we leave the try clause, regardless of whether we leave it:
  //   1) normally, after reaching the bottom of the block;
  //   2) because of a break, continue, or return statement;
  //   3) with an exception that is handled by a catch clause above; or
  //   4) with an uncaught exception that has not been handled.
  // If the try clause calls System.exit(), however, the interpreter
  // exits before the finally clause can be run.
}
```

2.6.16.1. try

The `try` clause simply establishes a block of code that either has its exceptions handled or needs special cleanup code to be run when it terminates for any reason. The `try` clause by itself doesn't do anything interesting; it is the `catch` and `finally` clauses that do the exception-handling and cleanup operations.

2.6.16.2. catch

A `try` block can be followed by zero or more `catch` clauses that specify code to handle various types of exceptions. Each `catch` clause is declared with a single argument that specifies the type of exceptions the clause can handle and also provides a name the clause can use to refer to the exception object it is currently handling. The type and name of an exception handled by a `catch` clause are exactly like the type and name of an argument passed to a method, except that for a `catch` clause, the argument type must be `Throwable` or one of its subclasses.

When an exception is thrown, the Java interpreter looks for a `catch` clause with an argument of the same type as the exception object or a superclass of that type. The interpreter invokes the first such `catch` clause it finds. The code within a

catch block should take whatever action is necessary to cope with the exceptional condition. If the exception is a java.io.FileNotFoundException exception, for example, you might handle it by asking the user to check his spelling and try again. It is not required to have a catch clause for every possible exception; in some cases the correct response is to allow the exception to propagate up and be caught by the invoking method. In other cases, such as a programming error signaled by NullPointerException, the correct response is probably not to catch the exception at all, but allow it to propagate and have the Java interpreter exit with a stack trace and an error message.

2.6.16.3. finally

The finally clause is generally used to clean up after the code in the try clause (e.g., close files, shut down network connections). What is useful about the finally clause is that it is guaranteed to be executed if any portion of the try block is executed, regardless of how the code in the try block completes. In fact, the only way a try clause can exit without allowing the finally clause to be executed is by invoking the System.exit() method, which causes the Java interpreter to stop running.

In the normal case, control reaches the end of the try block and then proceeds to the finally block, which performs any necessary cleanup. If control leaves the try block because of a return, continue, or break statement, the finally block is executed before control transfers to its new destination.

If an exception occurs in the try block, and there is an associated catch block to handle the exception, control transfers first to the catch block and then to the finally block. If there is no local catch block to handle the exception, control transfers first to the finally block, and then propagates up to the nearest containing catch clause that can handle the exception.

If a finally block itself transfers control with a return, continue, break, or throw statement or by calling a method that throws an exception, the pending control transfer is abandoned, and this new transfer is processed. For example, if a finally clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a finally clause issues a return statement, the method returns normally, even if an exception has been thrown and has not been handled yet.

try and finally can be used together without exceptions or any catch clauses. In this case, the finally block is simply cleanup code that is guaranteed to be executed, regardless of any break, continue, or return statements within the try clause.

In previous discussions of the for and continue statements, we've seen that a for loop cannot be naively translated into a while loop because the continue statement behaves slightly differently when used in a for loop than it does when used in a while loop. The finally clause gives us a way to write a while loop that is truly equivalent to a for loop. Consider the following generalized for loop:

```
for( initialize ; test ; increment )
    statement
```

The following while loop behaves the same, even if the statement block contains a continue statement:

```
initialize ;
while ( test ) {
  try { statement }
  finally { increment ; }
}
```

2.7. Methods

A *method* is a named collection of Java statements that can be invoked by other Java code. When a method is invoked, it is passed zero or more values known as arguments. The method performs some computations and, optionally, returns a value. A method invocation is an expression that is evaluated by the Java interpreter. Because method invocations can have side effects, however, they can also be used as expression statements.

You already know how to define the body of a method; it is simply an arbitrary sequence of statements enclosed within curly braces. What is more interesting about a method is its *signature*. The signature specifies:

- The name of the method

- The type and name of each of the parameters used by the method

- The type of the value returned by the method

- The exception types the method can throw

- Various method modifiers that provide additional information about the method

A method signature defines everything you need to know about a method before calling it. It is the method *specification* and defines the API for the method. The reference section of this book is essentially a list of method signatures for all publicly accessible methods of all publicly accessible classes of the Java platform. In order to use the reference section of this book, you need to know how to read a method signature. And, in order to write Java programs, you need to know how to define your own methods, each of which begins with a method signature.

A method signature looks like this:

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

The signature (the method specification) is followed by the method body (the method implementation), which is simply a sequence of Java statements enclosed in curly braces. In certain cases (described in [Chapter 3, "Object-Oriented Programming in Java"](#)), the implementation is omitted, and the method body is replaced with a single semicolon.

Here are some example method definitions. The method bodies have been omitted, so we can focus on the signatures:

```
public static void main(String[] args) { ... }
public final synchronized int indexOf(Object element, int startIndex) { ... }
double distanceFromOrigin() { ... }
static double squareRoot(double x) throws IllegalArgumentException { ... }
protected abstract String readText(File f, String encoding)
```

```
                    throws FileNotFoundException, UnsupportedEncodingException;
```

modifiers is zero or more special modifier keywords, separated from each other by spaces. A method might be declared with the `public` and `static` modifiers, for example. Other valid method modifiers are `abstract`, `final`, `native`, `private`, `protected`, and `synchronized`. The meanings of these modifiers are not important here; they are discussed in Chapter 3, "Object-Oriented Programming in Java".

The *type* in a method signature specifies the return type of the method. If the method returns a value, this is the name of a primitive type, an array type, or a class. If the method does not return a value, *type* must be `void`. A *constructor* is a special type of method used to initialize newly created objects. As we'll see in Chapter 3, "Object-Oriented Programming in Java", constructors are defined just like methods, except that their signatures do not include this *type* specification.

The *name* of a method follows the specification of its modifiers and type. Method names, like variable names, are Java identifiers and, like all Java identifiers, can use any characters of the Unicode character set. It is legal (and sometimes useful) to define more than one method with the same name, as long as each version of the method has a different parameter list. Defining multiple methods with the same name is called *method overloading*. The `System.out.println()` method we've seen so much of is an overloaded method. There is one method by this name that prints a string and other methods by the same name that print the values of the various primitive types. The Java compiler decides which method to call based on the type of the argument passed to the method.

When you are defining a method, the name of the method is always followed by the method's parameter list, which must be enclosed in parentheses. The parameter list defines zero or more arguments that are passed to the method. The parameter specifications, if there are any, each consist of a type and a name and are separated from each other by commas (if there are multiple parameters). When a method is invoked, the argument values it is passed must match the number, type, and order of the parameters specified in this method signature line. The values passed need not have exactly the same type as specified in the signature, but they must be convertible to those types without casting. C and C++ programmers should note that when a Java method expects no arguments, its parameter list is simply `()`, not `(void)`.

The final part of a method signature is the `throws` clause, which I first described when we discussed the `throw` statement. If a method uses the `throw` statement to throw a checked exception, or if it calls some other method that throws a checked exception and does not catch or handle that exception, the method must declare that it can throw that exception. If a method can throw one or more checked exceptions, it specifies this by placing the `throws` keyword after the argument list and following it by the name of the exception class or classes it can throw. If a method does not throw any exceptions, it does not use the `throws` keyword. If a method throws more than one type of exception, separate the names of the exception classes from each other with commas.

Chapter 2: Java Syntax from
the Ground Up

2.8. Classes and Objects

Now that we have introduced operators, expressions, statements, and methods, we can finally talk about classes. A *class* is a named collection of fields that hold data values and methods that operate on those values. Some classes also contain nested inner classes. Classes are the most fundamental structural element of all Java programs. You cannot write Java code without defining a class. All Java statements appear within methods, and all methods are defined within classes.

Classes are more than just another structural level of Java syntax. Just as a cell is the smallest unit of life that can survive and reproduce on its own, a class is the smallest unit of Java code that can stand alone. The Java compiler and interpreter do not recognize fragments of Java code that are smaller than a class. A class is the basic unit of execution for Java, which makes classes very important. Java actually defines another construct, called an *interface*, that is quite similar to a class. The distinction between classes and interfaces will become clear in Chapter 3, "Object-Oriented Programming in Java", but for now I'll use the term "class" to mean either a class or an interface.

Classes are important for another reason: every class defines a new data type. For example, you can define a class named `Point` to represent a data point in the two-dimensional Cartesian coordinate system. This class can define fields (each of type `double`) to hold the X and Y coordinates of a point and methods to manipulate and operate on the point. The `Point` class is a new data type.

When discussing data types, it is important to distinguish between the data type itself and the values the data type represents. `char` is a data type: it represents Unicode characters. But a `char` value represents a single specific character. A class is a data type; the value of a class type is called an *object*. We use the name class because each class defines a type (or kind, or species, or class) of objects. The `Point` class is a data type that represents X,Y points, while a `Point` object represents a single specific X,Y point. As you might imagine, classes and their objects are closely linked. In the sections that follow, we will be discussing both.

2.8.1. Defining a Class

Here is a possible definition of the `Point` class we have been discussing:

```
/** Represents a Cartesian (x,y) point */
public class Point {
  public double x, y;                  // The coordinates of the point.
  public Point(double x, double y) {   // A constructor that
    this.x = x; this.y = y;            // initializes the fields.
  }

  public double distanceFromOrigin() { // A method that operates on
    return Math.sqrt(x*x + y*y);       // the x and y fields.
  }
}
```

This class definition is stored in a file named *Point.java* and compiled to a file named *Point.class*, at which point it is available for use by Java programs and other classes. This class definition is provided here for completeness and to provide context, but don't expect to understand all the details just yet; most of [Chapter 3, "Object-Oriented Programming in Java"](#) is devoted to the topic of defining classes. Do pay extra attention to the first (non-comment) line of the class definition, however. Just as the first line of a method definition--the method signature--defines the API for the method, this line defines the basic API for a class (as described in the next chapter).

Keep in mind that you don't have to define every class you want to use in a Java program. The Java platform consists of over 1500 predefined classes that are guaranteed to be available on every computer that runs Java.

2.8.2. Creating an Object

Now that we have defined the `Point` class as a new data type, we can use the following line to declare a variable that holds a `Point` object:

```
Point p;
```

Declaring a variable to hold a `Point` object does not create the object itself, however. To actually create an object, you must use the `new` operator. This keyword is followed by the object's class (i.e., its type) and an optional argument list in parentheses. These arguments are passed to the constructor method for the class, which initializes internal fields in the new object:

```
// Create a Point object representing (2,-3.5) and store it in variable p
Point p = new Point(2.0, -3.5);

// Create some other objects as well
Date d = new Date();          // A Date object that represents the current time
Vector list = new Vector();  // A Vector object to hold a list of objects
```

The `new` keyword is by far the most common way to create objects in Java. There are a few other ways that are worth mentioning, however. First, there are a couple of classes that are so important that the Java language defines special literal syntax for creating objects of those types (as we'll discuss in the next section). Second, Java supports a dynamic loading mechanism that allows programs to load classes and create instances of those classes dynamically. This dynamic instantiation is done with the `newInstance()` methods of `java.lang.Class` and `java.lang.Constructor`. Finally, in Java 1.1 and later, objects can also be created by deserializing them. In other words, an object that has had its state saved, or serialized, usually to a file, can be recreated using the `java.io.ObjectInputStream` class.

2.8.3. Object Literals

As I just said, Java defines special syntax for creating instances of two very important classes. The first class is `String`, which represents text as a string of characters. Since programs usually communicate with their users through the written word, the ability to manipulate strings of text is quite important in any programming language. In some languages, strings are a primitive type, on a par with integers and characters. In Java, however, strings are objects; the data type used to represent text is the `String` class.

Because strings are such a fundamental data type, Java allows you to include text literally in programs by placing it between double-quote (`"`) characters. For example:

```
String name = "David";
```

```
System.out.println("Hello, " + name);
```

Don't confuse the double-quote characters that surround string literals with the single-quote (or apostrophe) characters that surround `char` literals. String literals can contain any of the escape sequences `char` literals can (see Table 2-3). Escape sequences are particularly useful for embedding double-quote characters within double-quoted string literals. For example:

```
String story = "\t\"How can you stand it?\" he asked sarcastically.\n";
```

String literals can be only a single line long. Java does not support any kind of continuation-character syntax that allows two separate lines to be treated as a single line. If you need to represent a long string of text that does not fit on a single line, break it into independent string literals and use the + operator to concatenate the literals. For example:

```
String s = "This is a test of the        // This is illegal; string literals
           emergency broadcast system";  // cannot be broken across lines.

String s = "This is a test of the " +    // Do this instead.
           "emergency broadcast system";
```

This concatenation of literals is done when your program is compiled, not when it is run, so you do not need to worry about any kind of performance penalty.

The second class that supports its own special object literal syntax is the class named `Class`. `Class` is a (self-referential) data type that represents all Java data types, including primitive types and array types, not just class types. To include a `Class` object literally in a Java program, follow the name of any data type with `.class`. For example:

```
Class typeInt = int.type;
Class typeIntArray = int[].type;
Class typePoint = Point.class;
```

This feature is supported by Java 1.1 and later.

The Java reserved word `null` is a special literal that can be used with any class. Instead of representing a literal object, it represents the absence of an object. For example:

```
String s = null;
Point p = null;
```

Finally, objects can also be included literally in a Java program through the use of a construct known as an anonymous inner class. Anonymous classes are discussed in Chapter 3, "Object-Oriented Programming in Java".

2.8.4. Using an Object

Now that we've seen how to define classes and instantiate them by creating objects, we need to look at the Java syntax that allows us to use those objects. Recall that a class defines a collection of fields and methods. Each object has its own copies of those fields and has access to those methods. We use the dot character (`.`) to access the named fields and methods of an object. For example:

```
Point p = new Point(2, 3);          // Create an object
```

```
    double x = p.x;                     // Read a field of the object
    p.y = p.x * p.x;                    // Set the value of a field
    double d = p.distanceFromOrigin();  // Access a method of the object
```

This syntax is central to object-oriented programming in Java, so you'll see it a lot. Note, in particular, the expression `p.distanceFromOrigin()`. This tells the Java compiler to look up a method named `distanceFromOrigin()` defined by the class `Point` and use that method to perform a computation on the fields of the object `p`. We'll cover the details of this operation in Chapter 3, "Object-Oriented Programming in Java".

← PREVIOUS	**HOME**	**NEXT →**
2.7. Methods	**BOOK INDEX**	2.9. Array Types

[Chapter 2: Java Syntax from the Ground Up](#)

2.9. Array Types

Array types are the second kind of reference types in Java. An array is an ordered collection, or numbered list, of values. The values can be primitive values, objects, or even other arrays, but all of the values in an array must be of the same type. The type of the array is the type of the values it holds, followed by the characters `[]`. For example:

```
byte b;                      // byte is a primitive type
byte[] arrayOfBytes;         // byte[] is an array type: array of byte
byte[][] arrayOfArrayOfBytes; // byte[][] is another type: array of byte[]
Point[] points;              // Point[] is an array of Point objects
```

For compatibility with C and C++, Java also supports another syntax for declaring variables of array type. In this syntax, one or more pairs of square brackets follow the name of the variable, rather than the name of the type:

```
byte arrayOfBytes[];           // Same as byte[] arrayOfBytes
byte arrayOfArrayOfBytes[][];  // Same as byte[][] arrayOfArrayOfBytes
byte[] arrayOfArrayOfBytes[];  // Ugh! Same as byte[][] arrayOfArrayOfBytes
```

This is almost always a confusing syntax, however, and it is not recommended.

With classes and objects, we have separate terms for the type and the values of that type. With arrays, the single word array does double duty as the name of both the type and the value. Thus, we can speak of the array type `int[]` (a type) and an array of `int` (a particular array value). In practice, it is usually clear from context whether a type or a value is being discussed.

2.9.1. Creating Arrays

To create an array value in Java, you use the `new` keyword, just as you do to create an object. Arrays don't need to be initialized like objects do, however, so you don't pass a list of arguments between parentheses. What you must specify, though, is how big you want the array to be. If you are creating a `byte[]`, for example, you must specify how many `byte` values you want it to hold. Array values have a fixed size in Java. Once an array is created, it can never grow or shrink. Specify the desired size of your array as a non-negative integer between square brackets:

```
byte[] buffer = new byte[1024];
String[] lines = new String[50];
```

When you create an array with this syntax, each of the values held in the array is automatically initialized to its default value. This is `false` for `boolean` values, `'\u0000'` for `char` values, 0 for integer values, 0.0 for floating-point values, and `null` for objects or array values.

2.9.2. Using Arrays

Once you've created an array with the `new` operator and the square-bracket syntax, you also use square brackets to access the individual values contained in the array. Remember that an array is an ordered collection of values. The elements of an array are numbered sequentially, starting with 0. The number of an array element refers to the element. This number is often called the *index*, and the process of looking up a numbered value in an array is sometimes called *indexing* the array.

To refer to a particular element of an array, simply place the index of the desired element in square brackets after the name of the array. For example:

```
String[] responses = new String[2];   // Create an array of two strings
responses[0] = "Yes";                  // Set the first element of the array
responses[1] = "No";                   // Set the second element of the array

// Now read these array elements
System.out.println(question + " (" + responses[0] + "/" +
                        responses[1] + " ): ");
```

In some programming languages, such as C and C++, it is a common bug to write code that tries to read or write array elements that are past the end of the array. Java does not allow this. Every time you access an array element, the Java interpreter automatically checks that the index you have specified is valid. If you specify a negative index or an index that is greater than the last index of the array, the interpreter throws an exception of type `ArrayIndexOutOfBoundsException`. This prevents you from reading or writing nonexistent array elements.

Array index values are integers; you cannot index an array with a floating-point value, a `boolean`, an object, or another array. `char` values can be converted to `int` values, so you *can* use characters as array indexes. Although `long` is an integer data type, `long` values cannot be used as array indexes. This may seem surprising at first, but consider that an `int` index supports arrays with over two billion elements. An `int[]` with this many elements would require eight gigabytes of memory. When you think of it this way, it is not surprising that `long` values are not allowed as array indexes.

Besides setting and reading the value of array elements, there is one other thing you can do with an array value. Recall that whenever we create an array, we must specify the number of elements the array holds. This value is referred to as the length of the array; it is an intrinsic property of the array. If you need to know the length of the array, append `.length` to the array name:

```
if (errorCode < errorMessages.length)
   System.out.println(errorMessages[errorCode]);
```

`.length` is special Java syntax for arrays. An expression like `a.length` looks as though it refers to a field of an object `a`, but this is not actually the case. The `.length` syntax can be used only to read the length of an array. It cannot be used to set the length of an array (because, in Java, an array has a fixed length that can never change).

In the previous example, the array index within square brackets is a variable, not an integer literal. In fact, arrays are most often used with loops, particularly `for` loops, where they are indexed using a variable that is incremented or decremented each time through the loop:

```
int[] values;                          // Array elements initialized elsewhere
int total = 0;                         // Store sum of elements here
for(int i = 0; i < values.length; i++) // Loop through array elements
   total += values[i];                 // Add them up
```

In Java, the first element of an array is always element number 0. If you are accustomed to a programming language that numbers array elements beginning with 1, this will take some getting used to. For an array `a`, the first element is `a[0]`, the second element is `a[1]`, and the last element is:

```
a[a.length - 1]          // The last element of any array named a
```

2.9.3. Array Literals

The `null` literal used to represent the absence of an object can also be used to represent the absence of an array. For example:

```
char[] password = null;
```

In addition to the `null` literal, Java also defines special syntax that allows you to specify array values literally in your programs. There are actually two different syntaxes for array literals. The first, and more commonly used, syntax can be used only when declaring a variable of array type. It combines the creation of the array object with the initialization of the array elements:

```
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

This creates an array that contains the eight `int` elements listed within the curly braces. Note that we don't use the `new` keyword or specify the type of the array in this array literal syntax. The type is implicit in the variable declaration of which the initializer is a part. Also, the array length is not specified explicitly with this syntax; it is determined implicitly by counting the number of elements listed between the curly braces. There is a semicolon following the close curly brace in this array literal. This is one of the fine points of Java syntax. When curly braces delimit classes, methods, and compound statements, they are not followed by semicolons. However, for this array literal syntax, the semicolon is required to terminate the variable declaration statement.

The problem with this array literal syntax is that it works only when you are declaring a variable of array type. Sometimes you need to do something with an array value (such as pass it to a method) but are going to use the array only once, so you don't want to bother assigning it to a variable. In Java 1.1 and later, there is an array literal syntax that supports this kind of anonymous arrays (so called because they are not assigned to variables, so they don't have names). This kind of array literal looks as follows:

```
// Call a method, passing an anonymous array literal that contains two strings
String response = askQuestion("Do you want to quit?",
                              new String[] {"Yes", "No"});

// Call another method with an anonymous array (of anonymous objects)
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
                                               new Point(3,4),
                                               new Point(3,2) });
```

With this syntax, you use the `new` keyword and specify the type of the array, but the length of the array is not explicitly specified.

It is important to understand that the Java Virtual Machine architecture does not support any kind of efficient array initialization. In other words, array literals are created and initialized when the program is run, not when the program is compiled. Consider the following array literal:

```
int[] perfectNumbers = {6, 28};
```

This is compiled into Java byte codes that are equivalent to:

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

Thus, if you want to include a large amount of data in a Java program, it may not be a good idea to include that data literally in an array, since the Java compiler has to create lots of Java byte codes to initialize the array, and then the Java interpreter has to laboriously execute all that initialization code. In cases like this, it is better to store your data in an external file and read it into the program at runtime.

The fact that Java does all array initialization explicitly at runtime has an important corollary, however. It means that the elements of an array literal can be arbitrary expressions that are computed at runtime, rather than constant expressions that are resolved by the compiler. For example:

```
Point[] points = { circle1.getCenterPoint(), circle2.getCenterPoint() };
```

2.9.4. Multidimensional Arrays

As we've seen, an array type is simply the element type followed by a pair of square brackets. An array of char is char[], and an array of arrays of char is char[][]. When the elements of an array are themselves arrays, we say that the array is *multidimensional*. In order to work with multidimensional arrays, there are a few additional details you must understand.

Imagine that you want to use a multidimensional array to represent a multiplication table:

```
int[][] products;      // A multiplication table
```

Each of the pairs of square brackets represents one dimension, so this is a two-dimensional array. To access a single int element of this two-dimensional array, you must specify two index values, one for each dimension. Assuming that this array was actually initialized as a multiplication table, the int value stored at any given element would be the product of the two indexes. That is, products[2][4] would be 8, and products[3][7] would be 21.

To create a new multidimensional array, use the new keyword and specify the size of both dimensions of the array. For example:

```
int[][] products = new int[10][10];
```

In some languages, an array like this would be created as a single block of 100 int values. Java does not work this way. This line of code does three things:

- Declares a variable named products to hold an array of arrays of int.

- Creates a 10-element array to hold 10 arrays of int.

- Creates 10 more arrays, each of which is a 10-element array of int. It assigns each of these 10 new arrays to the

elements of the initial array. The default value of every `int` element of each of these 10 new arrays is 0.

To put this another way, the previous single line of code is equivalent to the following code:

```
int[][] products = new int[10][];    // An array to hold ten int[] values.
for(int i = 0; i < 10; i++)          // Loop ten times...
  products[i] = new int[10];         // ...and create ten arrays.
```

The `new` keyword performs this additional initialization automatically for you. It works with arrays with more than two dimensions as well:

```
float[][][] globalTemperatureData = new float[360][180][100];
```

When using `new` with multidimensional arrays, you do not have to specify a size for all dimensions of the array, only the leftmost dimension or dimensions. For example, the following two lines are legal:

```
float[][][] globalTemperatureData = new float[360][][];
float[][][] globalTemperatureData = new float[360][180][];
```

The first line creates a single-dimensional array, where each element of the array can hold a `float[][]`. The second line creates a two-dimensional array, where each element of the array is a `float[]`. If you specify a size for only some of the dimensions of an array, however, those dimensions must be the leftmost ones. The following lines are not legal:

```
float[][][] globalTemperatureData = new float[360][][100];  // Error!
float[][][] globalTemperatureData = new float[][180][100];  // Error!
```

Like a one-dimensional array, a multidimensional array can be initialized using an array literal. Simply use nested sets of curly braces to nest arrays within arrays. For example, we can declare, create, and initialize a 5×5 multiplication table like this:

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

Or, if you want to use a multidimensional array without declaring a variable, you can use the anonymous initializer syntax:

```
boolean response = bilingualQuestion(question, new String[][] {
                                          { "Yes", "No" },
                                          { "Oui", "Non" }});
```

When you create a multidimensional array using the `new` keyword, you always get a *rectangular* array: one in which all the array values for a given dimension have the same size. This is perfect for rectangular data structures, such as matrixes. However, because multidimensional arrays are implemented as arrays of arrays in Java, instead of as a single rectangular block of elements, you are in no way constrained to use rectangular arrays. For example, since our multiplication table is symmetrical about the diagonal from top left to bottom right, we can represent the same information in a nonrectangular array with fewer elements:

```
int[][] products = { {0},
                     {0, 1},
                     {0, 2, 4},

                     {0, 3, 6, 9},
                     {0, 4, 8, 12, 16} };
```

When working with multidimensional arrays, you'll often find yourself using nested loops to create or initialize them. For example, you can create and initialize a large triangular multiplication table as follows:

```
int[][] products = new int[12][];         // An array of 12 arrays of int.
for(int row = 0; row < 12; row++) {        // For each element of that array,
  products[row] = new int[row+1];          // allocate an array of int.
  for(int col = 0; col < row+1; col++)     // For each element of the int[],
    products[row][col] = row * col;        // initialize it to the product.
}
```

Chapter 2: Java Syntax from
the Ground Up

2.10. Reference Types

Now that we have discussed the syntax for working with objects and arrays, we can return to the issue of why classes and array types are known as reference types. As we saw in Table 2-2, all the Java primitive types have well-defined standard sizes, so all primitive values can be stored in a fixed amount of memory (between one and eight bytes, depending on the type). But classes and array types are composite types; objects and arrays contain other values, so they do not have a standard size, and they often require quite a bit more memory than eight bytes. For this reason, Java does not manipulate objects and arrays directly. Instead, it manipulates *references* to objects and arrays. Because Java handles objects and arrays by reference, classes and array types are known as reference types. In contrast, Java handles values of the primitive types directly, or by value.

A reference to an object or an array is simply some fixed-size value that refers to the object or array in some way.[4] When you assign an object or array to a variable, you are actually setting the variable to hold a reference to that object or array. Similarly, when you pass an object or array to a method, what really happens is that the method is given a reference to the object or array through which it can manipulate the object or array.

[4] Typically, a reference is the memory address at which the object or array is stored. However, since Java references are opaque and cannot be manipulated in any way, this is an implementation detail.

C and C++ programmers should note that Java does not support the & address-of operator or the * and −> dereference operators. In Java, primitive types are always handled exclusively by value, and objects and arrays are always handled exclusively by reference. Furthermore, unlike pointers in C and C++, references in Java are entirely opaque: they cannot be converted to or from integers, and they cannot be incremented or decremented.

Although references are an important part of how Java works, Java programs cannot manipulate references in any way. Despite this, there are significant differences between the behavior of primitive types and reference types in two important areas: the way values are copied and the way they are compared for equality.

2.10.1. Copying Objects and Arrays

Consider the following code that manipulate a primitive int value:

```
int x = 42;
int y = x;
```

After these lines execute, the variable y contains a copy of the value held in the variable x. Inside the Java VM, there are two independent copies of the 32-bit integer 42.

Now think about what happens if we run the same basic code but use a reference type instead of a primitive type:

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

After this code runs, the variable q holds a copy of the reference held in the variable p. There is still only one copy of the Point object in the VM, but there are now two copies of the reference to that object. This has some important implications. Suppose the two previous lines of code are followed by this code:

```
System.out.println(p.x);  // Print out the X coordinate of p: 1.0
q.x = 13.0;               // Now change the X coordinate of q
System.out.println(p.x);  // Print out p.x again; this time it is 13.0
```

Since the variables p and q hold references to the same object, either variable can be used to make changes to the object, and those changes are visible through the other variable as well.

This behavior is not specific to objects; the same thing happens with arrays, as illustrated by the following code:

```
char[] greet = { 'h','e','l','l','o' };  // greet holds an array reference
char[] cuss = greet;                     // cuss holds the same reference
cuss[4] = '!';                           // Use reference to change an element
System.out.println(greet);               // Prints "hell!"
```

A similar difference in behavior between primitive types and reference types occurs when arguments are passed to methods. Consider the following method:

```
void changePrimitive(int x) {
  while(x > 0)
    System.out.println(x--);
}
```

When this method is invoked, the method is given a copy of the argument used to invoke the method in the parameter x. The code in the method uses x as a loop counter and decrements it to zero. Since x is a primitive type, the method has its own private copy of this value, so this is a perfectly reasonable thing to do.

On the other hand, consider what happens if we modify the method so that the parameter is a reference type:

```
void changeReference(Point p) {
  while(p.x > 0)
    System.out.println(p.x--);
}
```

When this method is invoked, it is passed a private copy of a reference to a Point object and can use this reference to change the Point object. Consider the following:

```
Point q = new Point(3.0, 4.5);  // A point with an X coordinate of 3
changeReference(q);             // Prints 3,2,1 and modifies the Point
System.out.println(q.x);        // The X coordinate of q is now 0!
```

When the changeReference() method is invoked, it is passed a copy of the reference held in variable q. Now both the variable q and the method parameter p hold references to the same object. The method can use its reference to change the contents of the object. Note, however, that it cannot change the contents of the variable q. In other words, the method can change the Point object beyond recognition, but it cannot change the fact that the variable q refers to that object.

The title of this section is "Copying Objects and Arrays," but, so far, we've only seen copies of references to objects and arrays, not copies of the objects and arrays themselves. To make an actual copy of an object or an array, you must use the special `clone()` method (inherited by all objects from `java.lang.Object`):

```
Point p = new Point(1,2);      // p refers to one object
Point q = (Point) p.clone();   // q refers to a copy of that object
q.y = 42;                       // Modify the copied object, but not the original

int[] data = {1,2,3,4,5};             // An array
int[] copy = (int[]) data.clone();    // A copy of the array
```

Note that a cast is necessary to coerce the return value of the `clone()` method to the correct type. The reason for this will become clear later in this chapter. There are a couple of points you should be aware of when using `clone()`. First, not all objects can be cloned. Java only allows an object to be cloned if the object's class has explicitly declared itself to be cloneable by implementing the `Cloneable` interface. (We haven't discussed interfaces or how they are implemented yet; that is covered in [Chapter 3, "Object-Oriented Programming in Java"](#).) The definition of `Point` that we showed earlier does not actually implement this interface, so our `Point` type, as implemented, is not cloneable. Note, however, that arrays are always cloneable. If you call the `clone()` method for a non-cloneable object, it throws a `CloneNotSupportedException`, so when you use the `clone()` method, you may want to use it within a `try` block to catch this exception.

The second thing you need to understand about `clone()` is that, by default, it is implemented to create a shallow copy of an object or array. The copied object or array contains copies of all the primitive values and references in the original object or array. In other words, any references in the object or array are copied, not cloned; `clone()` does not recursively make copies of the objects or arrays referred to by those references. A class may need to override this shallow copy behavior by defining its own version of the `clone()` method that explicitly performs a deeper copy where needed. To understand the shallow copy behavior of `clone()`, consider cloning a two-dimensional array of arrays:

```
int[][] data = {{1,2,3}, {4,5}};            // An array of 2 references
int[][] copy = (int[][]) data.clone();  // Copy the 2 refs to a new array
copy[0][0] = 99;                            // This changes data[0][0] too!
copy[1] = new int[] {7,8,9};                // This does not change data[1]
```

If you want to make a deep copy of this multidimensional array, you have to copy each dimension explicitly:

```
int[][] data = {{1,2,3}, {4,5}};            // An array of 2 references
int[][] copy = new int[data.length][]; // A new array to hold copied arrays
for(int i = 0; i < data.length; i++)
  copy[i] = (int[]) data[i].clone();
```

2.10.2. Comparing Objects and Arrays

We've seen that primitive types and reference types differ significantly in the way they are assigned to variables, passed to methods, and copied. The types also differ in the way they are compared for equality. When used with primitive values, the equality operator (= =) simply tests whether two values are identical (i.e., whether they have exactly the same bits). With reference types, however, = = compares references, not actual objects or arrays. In other words, = = tests whether two references refer to the same object or array; it does not test whether two objects or arrays have the same content. For example:

```
    String letter = "o";
    String s = "hello";                     // These two String objects
    String t = "hell" + letter;             // contain exactly the same text.
    if (s == t) System.out.println("equal"); // But they are not equal!

    byte[] a = { 1, 2, 3 };                 // An array.
    byte[] b = (byte[]) a.clone();          // A copy with identical content.
    if (a == b) System.out.println("equal"); // But they are not equal!
```

When working with reference types, there are two kinds of equality: equality of reference and equality of object. It is important to distinguish between these two kinds of equality. One way to do this is to use the word "equals" when talking about equality of references and the word "equivalent" when talking about two distinct object or arrays that have the same contents. Unfortunately, the designers of Java didn't use this nomenclature, as the method for testing whether one object is equivalent to another is named `equals()`. To test two objects for equivalence, pass one of them to the `equals()` method of the other:

```
    String letter = "o";
    String s = "hello";                     // These two String objects
    String t = "hell" + letter;             // contain exactly the same text.
    if (s.equals(t))                        // And the equals() method
      System.out.println("equivalent");     // tells us so.
```

All objects inherit an `equals()` method (from `Object`, but the default implementation simply uses = = to test for equality of references, not equivalence of content. A class that wants to allow objects to be compared for equivalence can define its own version of the `equals()` method. Our `Point` class does not do this, but the `String` class does, as indicated by the code above. You can call the `equals()` method on an array, but it is the same as using the = = operator, because arrays always inherit the default `equals()` method that compares references rather than array content. Starting in Java 1.2, you can compare arrays for equivalence with the convenience method `java.util.Arrays.equals()`. Prior to Java 1.2, however, you must loop through the elements of the arrays and compare them yourself.

2.10.3. The null Reference

We've seen the `null` keyword in our discussions of objects and arrays. Now that we have described references, it is worth revisiting `null` to point out that it is a special value that is a reference to nothing, or an absence of a reference. The default value for all reference types is `null`. The `null` value is unique in that it can be assigned to a variable of any reference type whatsoever.

2.10.4. Terminology: Pass by Value

I've said that Java handles arrays and objects "by reference." Don't confuse this with the phrase "pass by reference."[5] "Pass by reference" is a term used to describe the method-calling conventions of some programming languages. In a pass-by-reference language, values--even primitive values--are not passed directly to methods. Instead, methods are always passed references to values. Thus, if the method modifies its parameters, those modifications are visible when the method returns, even for primitive types.

[5]Unfortunately, previous editions of this book may have contributed to the confusion!

Java does *not* do this; it is a "pass by value" language. However, when a reference type is involved, the value that is passed is a reference. But this is not the same as pass-by-reference. If Java were a pass-by-reference language, when a reference type

was passed to a method, it would be passed as a reference to the reference.

2.10.5. Memory Allocation and Garbage Collection

As we've already noted, objects and arrays are composite values that can contain a number of other values and may require a substantial amount of memory. When you use the `new` keyword to create a new object or array or use an object or array literal in your program, Java automatically creates the object for you, allocating whatever amount of memory is necessary. You don't need to do anything to make this happen.

In addition, Java also automatically reclaims that memory for reuse when it is no longer needed. It does this through a process called *garbage collection*. An object is considered garbage when there are no longer any references to it stored in any variables, the fields of any objects, or the elements of any arrays. For example:

```
Point p = new Point(1,2);           // Create an object
double d = p.distanceFromOrigin();  // Use it for something
p = new Point(2,3);                 // Create a new object
```

After the Java interpreter executes the third line, a reference to the new `Point` object has replaced the reference to the first one. There are now no remaining references to the first object, so it is garbage. At some point, the garbage collector will discover this and reclaim the memory used by the object.

C programmers, who are used to using `malloc()` and `free()` to manage memory, and C++ programmers, who are used to explicitly deleting their objects with `delete`, may find it a little hard to relinquish control and trust the garbage collector. Even though it seems like magic, it really works! There is a slight performance penalty due to the use of garbage collection, and Java programs may sometimes slow down noticeably while the garbage collector is actively reclaiming memory. However, having garbage collection built into the language dramatically reduces the occurrence of memory leaks and related bugs and almost always improves programmer productivity.

2.10.6. Reference Type Conversions

When we discussed primitive types earlier in this chapter, we saw that values of certain types can be converted to values of other types. Widening conversions are performed automatically by the Java interpreter, as necessary. Narrowing conversions, however, can result in lost data, so the interpreter does not perform them unless explicitly directed to do so with a cast.

Java does not allow any kind of conversion from primitive types to reference types or vice versa. Java does allow widening and narrowing conversions among certain reference types, however. As we've seen, there are an infinite number of potential reference types. In order to understand the conversions that can be performed among these types, you need to understand that the types form a hierarchy, usually called the *class hierarchy*.

Every Java class *extends* some other class, known as its *superclass*. A class inherits the fields and methods of its superclass and then defines its own additional fields and methods. There is a special class named `Object` that serves as the root of the class hierarchy in Java. It does not extend any class, but all other Java classes extend `Object` or some other class that has `Object` as one of its ancestors. The `Object` class defines a number of special methods that are inherited (or overridden) by all classes. These include the `toString()`, `clone()`, and `equals()` methods described earlier.

The predefined `String` class and the `Point` class we defined earlier in this chapter both extend `Object`. Thus, we can say that all `String` objects are also `Object` objects. We can also say that all `Point` objects are `Object` objects. The opposite is not true, however. We cannot say that every `Object` is a `String` because, as we've just seen, some `Object`

objects are `Point` objects.

With this simple understanding of the class hierarchy, we can return to the rules of reference type conversion:

- An object cannot be converted to an unrelated type. The Java compiler does not allow you to convert a `String` to a `Point`, for example, even if you use a cast operator.

- An object can be converted to the type of a superclass. This is a widening conversion, so no cast is required. For example, a `String` value can be assigned to a variable of type `Object` or passed to a method where an `Object` parameter is expected. Note that no conversion is actually performed; the object is simply treated as if it were an instance of the superclass.

- An object can be converted to the type of a subclass, but this is a narrowing conversion and requires a cast. The Java compiler provisionally allows this kind of conversion, but the Java interpreter checks at runtime to make sure it is valid. Only cast an object to the type of a subclass if you are sure, based on the logic of your program, that the object is actually an instance of the subclass. If it is not, the interpreter throws a `ClassCastException`. For example, if we assign a `String` object to a variable of type `Object`, we can later cast the value of that variable back to type `String`:

  ```
  Object o = "string";     // Widening conversion from String to Object
  // Later in the program...
  String s = (String) o;  // Narrowing conversion from Object to String
  ```

- All array types are distinct, so an array of one type cannot be converted to an array of another type, even if the individual elements could be converted. For example, although a `byte` can be widened to an `int`, a `byte[]` cannot be converted to an `int[]`, even with an explicit cast.

- Arrays do not have a type hierarchy, but all arrays are considered instances of `Object`, so any array can be converted to an `Object` value through a widening conversion. A narrowing conversion with a cast can convert such an object value back to an array. For example:

  ```
  Object o = new int[] {1,2,3};  // Widening conversion from array to Object
  // Later in the program...
  int[] a = (int[]) o;           // Narrowing conversion back to array type
  ```

2.11. Packages and the Java Namespace

A *package* is a named collection of classes (and possibly subpackages). Packages serve to group related classes and define a namespace for the classes they contain.

The Java platform includes packages with names that begin with `java`, `javax`, and `org.omg`. (Sun also defines standard extensions to the Java platform in packages whose names begin with `javax`.) The most fundamental classes of the language are in the package `java.lang`. Various utility classes are in `java.util`. Classes for input and output are in `java.io`, and classes for networking are in `java.net`. Some of these packages contain subpackages. For example, `java.lang` contains two more specialized packages, named `java.lang.reflect` and `java.lang.ref`, and `java.util` contains a subpackage, `java.util.zip`, that contains classes for working with compressed ZIP archives.

Every class has both a simple name, which is the name given to it in its definition, and a fully qualified name, which includes the name of the package of which it is a part. The `String` class, for example, is part of the `java.lang` package, so its fully qualified name is `java.lang.String`.

2.11.1. Defining a Package

To specify the package a class is to be part of, you use a `package` directive. The `package` keyword, if it appears, must be the first token of Java code (i.e., the first thing other than comments and space) in the Java file. The keyword should be followed by the name of the desired package and a semicolon. Consider a file of Java code that begins with this directive:

```
package com.davidflanagan.jude;
```

All classes defined by this file are part of the package named `com.davidflanagan.jude`.

If no `package` directive appears in a file of Java code, all classes defined in that file are part of a default unnamed package. As we'll see in Chapter 3, "Object-Oriented Programming in Java", classes in the same package have special access to each other. Thus, except when you are writing simple example programs, you should always use the `package` directive to prevent access to your classes from totally unrelated classes that also just happen to be stored in the unnamed package.

2.11.2. Importing Classes and Packages

A class in a package `p` can refer to any other class in `p` by its simple name. And, since the classes in the `java.lang` package are so fundamental to the Java language, any Java code can refer to any class in this package by its simple name. Thus, you can always type `String`, instead of `java.lang.String`. By default, however, you must use the fully qualified name of all other classes. So, if you want to use the `File` class of the `java.io` package, you must type `java.io.File`.

Specifying package names explicitly all the time quickly gets tiring, so Java includes an `import` directive you can use to save some typing. `import` is used to specify classes and packages of classes that can be referred to by their simple names instead of by their fully qualified names. The `import` keyword can be used any number of times in a Java file, but all uses must be at the top of the file, immediately after the `package` directive, if there is one. There can be comments between the `package` directive and the `import` directives, of course, but there cannot be any other Java code.

The `import` directive is available in two forms. To specify a single class that can be referred to by its simple name, follow the `import` keyword with the name of the class and a semicolon:

```
import java.io.File;    // Now we can type File instead of java.io.File
```

To import an entire package of classes, follow `import` with the name of the package, the characters `.*`, and a semicolon. Thus, if you want to use several other classes from the `java.io` package in addition to the `File` class, you can simply import the entire package:

```
import java.io.*;   // Now we can use simple names for all classes in java.io
```

This package `import` syntax does not apply to subpackages. If I import the `java.util` package, I must still refer to the `java.util.zip.ZipInputStream` class by its fully qualified name. If two classes with the same name are both imported from different packages, neither one can be referred to by its simple name; to resolve this naming conflict unambiguously, you must use the fully qualified name of both classes.

2.11.3. Globally Unique Package Names

One of the important functions of packages is to partition the Java namespace and prevent name collisions between classes. It is only their package names that keep the `java.util.List` and `java.awt.List` classes distinct, for example. In order for this to work, however, package names must themselves be distinct. As the developer of Java, Sun controls all package names that begin with `java`, `javax`, and `sun`.

For the rest of us, Sun proposes a package-naming scheme, which, if followed correctly, guarantees globally unique package names. The scheme is to use your Internet domain name, with its elements reversed, as the prefix for all your package names. My web site is *davidflanagan.com*, so all my Java packages begin with `com.davidflanagan`. It is up to me to decide how to partition the namespace below `com.davidflanagan`, but since I own that domain name, no other person or organization who is playing by the rules can define a package with the same name as any of mine.

[Chapter 2: Java Syntax from the Ground Up](#)

2.12. Java File Structure

This chapter has taken us from the smallest to the largest elements of Java syntax, from individual characters and tokens to operators, expressions, statements, and methods, and on up to classes and packages. From a practical standpoint, the unit of Java program structure you will be dealing with most often is the Java file. A Java file is the smallest unit of Java code that can be compiled by the Java compiler. A Java file consists of:

- An optional `package` directive

- Zero or more `import` directives

- One or more class definitions

These elements can be interspersed with comments, of course, but they must appear in this order. This is all there is to a Java file. All Java statements (except the `package` and `import` directives, which are not true statements) must appear within methods, and all methods must appear within a class definition.

There are a couple of other important restrictions on Java files. First, each file can contain at most one class that is declared `public`. A `public` class is one that is designed for use by other classes in other packages. We'll talk more about `public` and related modifiers in [Chapter 3, "Object-Oriented Programming in Java"](#). This restriction on public classes only applies to top-level classes; a class can contain any number of nested or inner classes that are declared `public`, as we'll see in [Chapter 3, "Object-Oriented Programming in Java"](#).

The second restriction concerns the filename of a Java file. If a Java file contains a `public` class, the name of the file must be the same as the name of the class, with the extension *.java* appended. Thus, if `Point` is defined as a `public` class, its source code must appear in a file named *Point.java*. Regardless of whether your classes are `public` or not, it is good programming practice to define only one per file and to give the file the same name as the class.

When a Java file is compiled, each of the classes it defines is compiled into a separate *class file* that contains Java byte codes to be interpreted by the Java Virtual Machine. A class file has the same name as the class it defines, with the extension *.class* appended. Thus, if the file *Point.java* defines a class named

Point, a Java compiler compiles it to a file named *Point.class*. On most systems, class files are stored in directories that correspond to their package names. Thus, the class com.davidflanagan.jude.DataFile is defined by the class file *com/davidflanagan/jude/DataFile.class*.

The Java interpreter knows where the class files for the standard system classes are located and can load them as needed. When the interpreter runs a program that wants to use a class named com.davidflanagan.jude.DataFile, it knows that the code for that class is located in a directory named *com/davidflanagan/jude* and, by default, it "looks" in the current directory for a subdirectory of that name. In order to tell the interpreter to look in locations other than the current directory, you must use the -classpath option when invoking the interpreter or set the CLASSPATH environment variable. For details, see the documentation for the Java interpreter, *java*, in Chapter 8, "Java Development Tools".

Chapter 2: Java Syntax from
the Ground Up

2.13. Defining and Running Java Programs

A Java program consists of a set of interacting class definitions. But not every Java class or Java file defines a program. To create a program, you must define a class that has a special method with the following signature:

```
public static void main(String args[])
```

This `main()` method is the main entry point for your program. It is where the Java interpreter starts running. This method is passed an array of strings and returns no value. When `main()` returns, the Java interpreter exits (unless `main()` has created separate threads, in which case the interpreter waits for all those threads to exit).

To run a Java program, you run the Java interpreter, *java*, specifying the fully qualified name of the class that contains the `main()` method. Note that you specify the name of the class, *not* the name of the class file that contains the class. Any additional arguments you specify on the command line are passed to the `main()` method as its `String[]` parameter. You may also need to specify the `-classpath` option to tell the interpreter to look for the classes needed by the program. Consider the following command:

```
C:\> java -classpath /usr/local/Jude com.davidflanagan.jude.Jude datafile.jude
```

java is the command to run the Java interpreter. *-classpath /usr/local/Jude* tells the interpreter where to look for *.class* files. `com.davidflanagan.jude.Jude` is the name of the program to run (i.e., the name of the class that defines the `main()` method). Finally, *datafile.jude* is a string that is passed to that `main()` method as the single element of an array of `String` objects.

In Java 1.2, there is an easier way to run programs. If a program and all its auxiliary classes (except those that are part of the Java platform) have been properly bundled in a Java archive (JAR) file, you can run the program simply by specifying the name of the JAR file:

```
C:\> java -jar /usr/local/Jude/jude.jar datafile.jude
```

Some operating systems make JAR files automatically executable. On those systems, you can simply say:

```
C:\> /usr/local/Jude/jude.jar datafile.jude
```

See Chapter 8, "Java Development Tools" for details.

2.12. Java File Structure 2.14. Differences Between C
and Java

2.14. Differences Between C and Java

If you are a C or C++ programmer, you should have found much of the syntax of Java--particularly at the level of operators and statements--to be familiar. Because Java and C are so similar in some ways, it is important for C and C++ programmers to understand where the similarities end. There are a number of important differences between C and Java, which are summarized in the following list:

No preprocessor

Java does not include a preprocessor and does not define any analogs of the `#define`, `#include`, and `#ifdef` directives. Constant definitions are replaced with `staticfinal` fields in Java. (See the `java.lang.Math.PI` field for an example.) Macro definitions are not available in Java, but advanced compiler technology and inlining has made them less useful. Java does not require an `#include` directive because Java has no header files. Java class files contain both the class API and the class implementation, and the compiler reads API information from class files as necessary. Java lacks any form of conditional compilation, but its cross-platform portability means that this feature is very rarely needed.

No global variables

Java defines a very clean namespace. Packages contain classes, classes contain fields and methods, and methods contain local variables. But there are no global variables in Java, and, thus, there is no possibility of namespace collisions among those variables.

Well-defined primitive type sizes

All the primitive types in Java have well-defined sizes. In C, the size of `short`, `int`, and `long` types is platform-dependent, which hampers portability.

No pointers

Java classes and arrays are reference types, and references to objects and arrays are akin to pointers in C. Unlike C pointers, however, references in Java are entirely opaque. There is no way to convert a reference to a primitive type, and a reference cannot be incremented or decremented.

There is no address-of operator like `&`, dereference operator like `*` or `->`, or `sizeof` operator. Pointers are a notorious source of bugs. Eliminating them simplifies the language and makes Java programs more robust and secure.

Garbage collection

The Java Virtual Machine performs garbage collection so that Java programmers do not have to explicitly manage the memory used by all objects and arrays. This feature eliminates another entire category of common bugs and all but eliminates memory leaks from Java programs.

No goto statement

Java doesn't support a `goto` statement. Use of `goto` except in certain well-defined circumstances is regarded as poor programming practice. Java adds exception handling and labeled `break` and `continue` statements to the flow-control statements offered by C. These are a good substitute for `goto`.

Variable declarations anywhere

C requires local variable declarations to be made at the beginning of a method or block, while Java allows them anywhere in a method or block. Many programmers prefer to keep all their variable declarations grouped together at the top of a method, however.

Forward references

The Java compiler is smarter than the C compiler, in that it allows methods to be invoked before they are defined. This eliminates the need to declare functions in a header file before defining them in a program file, as is done in C.

Method overloading

Java programs can define multiple methods with the same name, as long as the methods have different parameter lists.

No struct and union types

Java doesn't support C `struct` and `union` types. A Java `class` can be thought of as an enhanced `struct`, however.

No enumerated types

Java doesn't support the `enum` keyword used in C to define types that consist of fixed sets of named values. This is surprising for a strongly typed language like Java, but there are ways to simulate this feature with object constants.

No bitfields

Java doesn't support the (infrequently used) ability of C to specify the number of individual bits occupied by fields of a `struct`.

No typedef

Java doesn't support the `typedef` keyword used in C to define aliases for type names. Java's lack of pointers makes its type-naming scheme simpler and more consistent than C's, however, so many of the common uses of `typedef` are not really necessary in Java.

No method pointers

C allows you to store the address of a function in a variable and pass this function pointer to other functions. You cannot do this with Java methods, but you can often achieve similar results by passing an object that implements a particular interface. Also, a Java method can be represented and invoked through a `java.lang.reflect.Method` object.

No variable-length argument lists

Java doesn't allow you to define methods such as C's `printf()` that take a variable number of arguments. Method overloading allows you to simulate C varargs functions for simple cases, but there's no general replacement for this feature.