Chapter 3

Fundamental Programming Structures in Java

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Big Numbers
- Arrays

At this point, we are assuming that you successfully installed the JDK and were able to run the sample programs that we showed you in Chapter 2. It's time to start programming. This chapter shows you how the basic programming concepts such as data types, branches, and loops are implemented in Java.

Unfortunately, in Java you can't easily write a program that uses a GUI—you need to learn a fair amount of machinery to put up windows, add text boxes and buttons that respond to them, and so on. Because introducing the techniques needed to write GUI-based Java programs would take us too far away from our goal of introducing the basic programming concepts, the sample programs in this chapter are "toy" programs, designed to illustrate a concept. All these examples simply use a shell window for input and output.

Finally, if you are an experienced C++ programmer, you can get away with just skimming this chapter: Concentrate on the C/C++ notes that are interspersed throughout the text. Programmers coming from another background, such as Visual Basic, will find most of the concepts familiar, but all of the syntax very different—you should read this chapter very carefully.

A Simple Java Program

Let's look more closely at about the simplest Java program you can have—one that simply prints a message to the console window:

```
public class FirstSample
{
   public static void main(String[] args)
   {
      System.out.println("We will not use 'Hello, World!'");
   }
}
```

It is worth spending all the time that you need to become comfortable with the framework of this sample; the pieces will recur in all applications. First and foremost, *Java is case sensitive*. If you made any mistakes in capitalization (such as typing `Main` instead of `main`), the program will not run.

Now let's look at this source code line by line. The keyword `public` is called an *access modifier*; these modifiers control the level of access other parts of a program have to this code. We have more to say about access modifiers in Chapter 5. The keyword `class` reminds you that everything in a Java program lives inside a class. Although we spend a lot more time on classes in the next chapter, for now think of a class as a container for the program logic that defines the behavior of an application. As mentioned in Chapter 1, classes are the building blocks with which all Java applications and applets are built. *Everything* in a Java program must be inside a class.

Following the keyword `class` is the name of the class. The rules for class names in Java are quite generous. Names must begin with a letter, and after that, they can have any combination of letters and digits. The length is essentially unlimited. You cannot use a Java reserved word (such as `public` or `class`) for a class name. (See the Appendix for a list of reserved words.)

The standard naming convention (which we follow in the name `FirstSample`) is that class names are nouns that start with an uppercase letter. If a name consists of multiple words, use an initial uppercase letter in each of the words. (This use of uppercase letters in the middle of a word is sometimes called "camel case" or, self-referentially, "CamelCase.")

You need to make the file name for the source code the same as the name of the public class, with the extension `.java` appended. Thus, you must store this code in a file called `FirstSample.java`. (Again, case is important—don't use `firstsample.java`.)

If you have named the file correctly and not made any typos in the source code, then when you compile this source code, you end up with a file containing the bytecodes for this class. The Java compiler automatically names the bytecode file `FirstSample.class` and stores it in the same directory as the source file. Finally, launch the program by issuing the following command:

```
java FirstSample
```

(Remember to leave off the `.class` extension.) When the program executes, it simply displays the string `We will not use 'Hello, World!'` on the console.

When you use

```
java ClassName
```

to run a compiled program, the Java virtual machine always starts execution with the code in the `main` method in the class you indicate. (The term "method" is Java-speak for a function.) Thus, you *must* have a `main` method in the source file for your class for your code to execute. You can, of course, add your own methods to a class and call them from the `main` method. (We cover writing your own methods in the next chapter.)

NOTE: According to the Java Language Specification, the `main` method must be declared `public`. (The Java Language Specification is the official document that describes the Java language. You can view or download it from `http://java.sun.com/docs/books/jls`.)

However, several versions of the Java launcher were willing to execute Java programs even when the `main` method was not `public`. A programmer filed a bug report. To see it, visit the site `http://bugs.sun.com/bugdatabase/index.jsp` and enter the bug identification number 4252539. That bug was marked as "closed, will not be fixed." A Sun engineer added an explanation that the Java Virtual Machine Specification (at `http://java.sun.com/docs/books/vmspec`) does not mandate that `main` is `public` and that "fixing it will cause potential troubles." Fortunately, sanity finally prevailed. The Java launcher in Java SE 1.4 and beyond enforces that the `main` method is `public`.

There are a couple of interesting aspects about this story. On the one hand, it is frustrating to have quality assurance engineers, who are often overworked and not always experts in the fine points of Java, make questionable decisions about bug reports. On the other hand, it is remarkable that Sun puts the bug reports and their resolutions onto the Web, for anyone to scrutinize. The "bug parade" is a very useful resource for programmers. You can even vote for your favorite bug. Bugs with lots of votes have a high chance of being fixed in the next JDK release.

Notice the braces { } in the source code. In Java, as in C/C++, braces delineate the parts (usually called *blocks*) in your program. In Java, the code for any method must be started by an opening brace { and ended by a closing brace }.

Brace styles have inspired an inordinate amount of useless controversy. We use a style that lines up matching braces. Because whitespace is irrelevant to the Java compiler, you can use whatever brace style you like. We will have more to say about the use of braces when we talk about the various kinds of loops.

For now, don't worry about the keywords `static void`—just think of them as part of what you need to get a Java program to compile. By the end of Chapter 4, you will understand this incantation completely. The point to remember for now is that every Java application must have a `main` method that is declared in the following way:

```
public class ClassName
{
   public static void main(String[] args)
   {
      program statements
   }
}
```

C++ NOTE: As a C++ programmer, you know what a class is. Java classes are similar to C++ classes, but there are a few differences that can trap you. For example, in Java *all* functions are methods of some class. (The standard terminology refers to them as methods, not member functions.) Thus, in Java you must have a shell class for the `main` method. You may also be familiar with the idea of *static member functions* in C++. These are member functions defined inside a class that do not operate on objects. The `main` method in Java is always static. Finally, as in C/C++, the `void` keyword indicates that this method does not return a value. Unlike C/C++, the `main` method does not return an "exit code" to the operating system. If the `main` method exits normally, the Java program has the exit code 0, indicating successful completion. To terminate the program with a different exit code, use the `System.exit` method.

Next, turn your attention to this fragment:

```
{
   System.out.println("We will not use 'Hello, World!'");
}
```

Braces mark the beginning and end of the *body* of the method. This method has only one statement in it. As with most programming languages, you can think of Java statements as being the sentences of the language. In Java, every statement must end with a semicolon. In particular, carriage returns do not mark the end of a statement, so statements can span multiple lines if need be.

The body of the `main` method contains a statement that outputs a single line of text to the console.

Here, we are using the `System.out` object and calling its `println` method. Notice the periods used to invoke a method. Java uses the general syntax

object.method(*parameters*)

for its equivalent of function calls.

In this case, we are calling the `println` method and passing it a string parameter. The method displays the string parameter on the console. It then terminates the output line so that each call to `println` displays its output on a new line. Notice that Java, like C/C++, uses double quotes to delimit strings. (You can find more information about strings later in this chapter.)

Methods in Java, like functions in any programming language, can use zero, one, or more *parameters* (some programmers call them *arguments*). Even if a method takes no parameters, you must still use empty parentheses. For example, a variant of the `println` method with no parameters just prints a blank line. You invoke it with the call

```
System.out.println();
```

NOTE: `System.out` also has a `print` method that doesn't add a new line character to the output. For example, `System.out.print("Hello")` prints `Hello` without a new line. The next output appears immediately after the letter `o`.

Comments

Comments in Java, like comments in most programming languages, do not show up in the executable program. Thus, you can add as many comments as needed without fear of bloating the code. Java has three ways of marking comments. The most common method is a //. You use this for a comment that will run from the // to the end of the line.

```
System.out.println("We will not use 'Hello, World!'"); // is this too cute?
```

When longer comments are needed, you can mark each line with a //. Or you can use the /* and */ comment delimiters that let you block off a longer comment. This is shown in Listing 3–1.

Listing 3–1 FirstSample.java

```
1.  /**
2.   * This is the first sample program in Core Java Chapter 3
3.   * @version 1.01 1997-03-22
4.   * @author Gary Cornell
5.   */
6.  public class FirstSample
7.  {
8.     public static void main(String[] args)
9.     {
10.       System.out.println("We will not use 'Hello, World!'");
11.    }
12. }
```

Finally, a third kind of comment can be used to generate documentation automatically. This comment uses a /** to start and a */ to end. For more on this type of comment and on automatic documentation generation, see Chapter 4.

CAUTION: /* */ comments do not nest in Java. That is, you cannot deactivate code simply by surrounding it with /* and */ because the code that you want to deactivate might itself contain a */ delimiter.

Data Types

Java is a *strongly typed language*. This means that every variable must have a declared type. There are eight *primitive types* in Java. Four of them are integer types; two are floating-point number types; one is the character type `char`, used for code units in the Unicode encoding scheme (see the section "The `char` Type" on page 42); and one is a `boolean` type for truth values.

NOTE: Java has an arbitrary precision arithmetic package. However, "big numbers," as they are called, are Java *objects* and not a new Java type. You see how to use them later in this chapter.

Integer Types

The integer types are for numbers without fractional parts. Negative values are allowed. Java provides the four integer types shown in Table 3–1.

Table 3–1 Java Integer Types

Type	Storage Requirement	Range (Inclusive)
`int`	4 bytes	–2,147,483,648 to 2,147,483, 647 (just over 2 billion)
`short`	2 bytes	–32,768 to 32,767
`long`	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`byte`	1 byte	–128 to 127

In most situations, the `int` type is the most practical. If you want to represent the number of inhabitants of our planet, you'll need to resort to a `long`. The `byte` and `short` types are mainly intended for specialized applications, such as low-level file handling, or for large arrays when storage space is at a premium.

Under Java, the ranges of the integer types do not depend on the machine on which you will be running the Java code. This alleviates a major pain for the programmer who wants to move software from one platform to another, or even between operating systems on the same platform. In contrast, C and C++ programs use the most efficient integer type for each processor. As a result, a C program that runs well on a 32-bit processor may exhibit integer overflow on a 16-bit system. Because Java programs must run with the same results on all machines, the ranges for the various types are fixed.

Long integer numbers have a suffix `L` (for example, `4000000000L`). Hexadecimal numbers have a prefix `0x` (for example, `0xCAFE`). Octal numbers have a prefix `0`. For example, `010` is 8. Naturally, this can be confusing, and we recommend against the use of octal constants.

C++ NOTE: In C and C++, `int` denotes the integer type that depends on the target machine. On a 16-bit processor, like the 8086, integers are 2 bytes. On a 32-bit processor like the Sun SPARC, they are 4-byte quantities. On an Intel Pentium, the integer type of C and C++ depends on the operating system: For DOS and Windows 3.1, integers are 2 bytes. When 32-bit mode is used for Windows programs, integers are 4 bytes. In Java, the sizes of all numeric types are platform independent.

Note that Java does not have any `unsigned` types.

Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in Table 3–2.

Table 3–2 Floating-Point Types

Type	Storage Requirement	Range
`float`	4 bytes	approximately ±3.40282347E+38F (6–7 significant decimal digits)
`double`	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)

The name `double` refers to the fact that these numbers have twice the precision of the `float` type. (Some people call these *double-precision* numbers.) Here, the type to choose in most applications is `double`. The limited precision of `float` is simply not sufficient for many situations. Seven significant (decimal) digits may be enough to precisely express your annual salary in dollars and cents, but it won't be enough for your company president's salary. The only reasons to use `float` are in the rare situations in which the slightly faster processing of single-precision numbers is important or when you need to store a large number of them.

Numbers of type `float` have a suffix `F` (for example, `3.402F`). Floating-point numbers without an `F` suffix (such as `3.402`) are always considered to be of type `double`. You can optionally supply the `D` suffix (for example, `3.402D`).

NOTE: As of Java SE 5.0, you can specify floating-point numbers in hexadecimal! For example, `0.125` = 2^{-3} can be written as `0x1.0p-3`. In hexadecimal notation, you use a `p`, not an `e`, to denote the exponent. Note that the mantissa is written in hexadecimal and the exponent in decimal. The base of the exponent is 2, not 10.

All floating-point computations follow the IEEE 754 specification. In particular, there are three special floating-point values to denote overflows and errors:

- Positive infinity
- Negative infinity
- NaN (not a number)

For example, the result of dividing a positive number by 0 is positive infinity. Computing 0/0 or the square root of a negative number yields NaN.

NOTE: The constants `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN` (as well as corresponding `Float` constants) represent these special values, but they are rarely used in practice. In particular, you cannot test

```
if (x == Double.NaN) // is never true
```

to check whether a particular result equals `Double.NaN`. All "not a number" values are considered distinct. However, you can use the `Double.isNaN` method:

```
if (Double.isNaN(x)) // check whether x is "not a number"
```

CAUTION: Floating-point numbers are *not* suitable for financial calculation in which roundoff errors cannot be tolerated. For example, the command `System.out.println(2.0 - 1.1)` prints `0.8999999999999999`, not `0.9` as you would expect. Such roundoff errors are caused by the fact that floating-point numbers are represented in the binary number system. There is no precise binary representation of the fraction 1/10, just as there is no accurate representation of the fraction 1/3 in the decimal system. If you need precise numerical computations without roundoff errors, use the `BigDecimal` class, which is introduced later in this chapter.

The `char` Type

The `char` type is used to describe individual characters. Most commonly, these will be character constants. For example, `'A'` is a character constant with value 65. It is different from `"A"`, a string containing a single character. Unicode code units can be expressed as hexadecimal values that run from `\u0000` to `\uFFFF`. For example, `\u2122` is the trademark symbol (™) and `\u03C0` is the Greek letter pi (π).

Besides the `\u` escape sequences that indicate the encoding of Unicode code units, there are several escape sequences for special characters, as shown in Table 3–3. You can use these escape sequences inside quoted character constants and strings, such as `'\u2122'` or `"Hello\n"`. The `\u` escape sequence (but none of the other escape sequences) can even be used *outside* quoted character constants and strings. For example,

```
public static void main(String\u005B\u005D args)
```

is perfectly legal—`\u005B` and `\u005D` are the encodings for [and].

Table 3–3 Escape Sequences for Special Characters

Escape Sequence	Name	Unicode Value
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000a
\r	Carriage return	\u000d

Table 3–3 Escape Sequences for Special Characters (continued)

Escape Sequence	Name	Unicode Value
\"	Double quote	\u0022
\'	Single quote	\u0027
\\	Backslash	\u005c

To fully understand the `char` type, you have to know about the Unicode encoding scheme. Unicode was invented to overcome the limitations of traditional character encoding schemes. Before Unicode, there were many different standards: ASCII in the United States, ISO 8859-1 for Western European languages, KOI-8 for Russian, GB18030 and BIG-5 for Chinese, and so on. This causes two problems. A particular code value corresponds to different letters in the various encoding schemes. Moreover, the encodings for languages with large character sets have variable length: Some common characters are encoded as single bytes, others require two or more bytes.

Unicode was designed to solve these problems. When the unification effort started in the 1980s, a fixed 2-byte width code was more than sufficient to encode all characters used in all languages in the world, with room to spare for future expansion—or so everyone thought at the time. In 1991, Unicode 1.0 was released, using slightly less than half of the available 65,536 code values. Java was designed from the ground up to use 16-bit Unicode characters, which was a major advance over other programming languages that used 8-bit characters.

Unfortunately, over time, the inevitable happened. Unicode grew beyond 65,536 characters, primarily due to the addition of a very large set of ideographs used for Chinese, Japanese, and Korean. Now, the 16-bit `char` type is insufficient to describe all Unicode characters.

We need a bit of terminology to explain how this problem is resolved in Java, beginning with Java SE 5.0. A *code point* is a code value that is associated with a character in an encoding scheme. In the Unicode standard, code points are written in hexadecimal and prefixed with U+, such as U+0041 for the code point of the letter A. Unicode has code points that are grouped into 17 *code planes*. The first code plane, called the *basic multilingual plane*, consists of the "classic" Unicode characters with code points U+0000 to U+FFFF. Sixteen additional planes, with code points U+10000 to U+10FFFF, hold the *supplementary characters*.

The UTF-16 encoding is a method of representing all Unicode code points in a variable-length code. The characters in the basic multilingual plane are represented as 16-bit values, called *code units*. The supplementary characters are encoded as consecutive pairs of code units. Each of the values in such an encoding pair falls into a range of 2048 unused values of the basic multilingual plane, called the *surrogates area* (U+D800 to U+DBFF for the first code unit, U+DC00 to U+DFFF for the second code unit). This is rather clever, because you can immediately tell whether a code unit encodes a single character or whether it is the first or second part of a supplementary character. For example, the mathematical symbol for the set of integers $\mathbb{Z}$ has code point U+1D56B

and is encoded by the two code units U+D835 and U+DD6B. (See http://en.wikipedia.org/wiki/UTF-16 for a description of the encoding algorithm.)

In Java, the char type describes a *code unit* in the UTF-16 encoding.

Our strong recommendation is not to use the char type in your programs unless you are actually manipulating UTF-16 code units. You are almost always better off treating strings (which we will discuss in the section "Strings" on page 53) as abstract data types.

The boolean Type

The boolean type has two values, false and true. It is used for evaluating logical conditions. You cannot convert between integers and boolean values.

C++ NOTE: In C++, numbers and even pointers can be used in place of boolean values. The value 0 is equivalent to the bool value false, and a non-zero value is equivalent to true. This is *not* the case in Java. Thus, Java programmers are shielded from accidents such as

```
if (x = 0) // oops...meant x == 0
```

In C++, this test compiles and runs, always evaluating to false. In Java, the test does not compile because the integer expression x = 0 cannot be converted to a boolean value.

Variables

In Java, every variable has a *type*. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

Notice the semicolon at the end of each declaration. The semicolon is necessary because a declaration is a complete Java statement.

A variable name must begin with a letter and must be a sequence of letters or digits. Note that the terms "letter" and "digit" are much broader in Java than in most languages. A letter is defined as 'A'-'Z', 'a'-'z', '_', or *any* Unicode character that denotes a letter in a language. For example, German users can use umlauts such as 'ä' in variable names; Greek speakers could use a π. Similarly, digits are '0'-'9' and *any* Unicode characters that denote a digit in a language. Symbols like '+' or '©' cannot be used inside variable names, nor can spaces. *All* characters in the name of a variable are significant and *case is also significant*. The length of a variable name is essentially unlimited.

TIP: If you are really curious as to what Unicode characters are "letters" as far as Java is concerned, you can use the isJavaIdentifierStart and isJavaIdentifierPart methods in the Character class to check.

You also cannot use a Java reserved word for a variable name. (See the Appendix for a list of reserved words.)

You can have multiple declarations on a single line:

```
int i, j; // both are integers
```

However, we don't recommend this style. If you declare each variable separately, your programs are easier to read.

NOTE: As you saw, names are case sensitive, for example, `hireday` and `hireDay` are two separate names. In general, you should not have two names that only differ in their letter case. However, sometimes it is difficult to come up with a good name for a variable. Many programmers then give the variable the same name of the type, such as

```
Box box; // ok--Box is the type and box is the variable name
```

Other programmers prefer to use an "a" prefix for the variable:

```
Box aBox;
```

Initializing Variables

After you declare a variable, you must explicitly initialize it by means of an assignment statement—you can never use the values of uninitialized variables. For example, the Java compiler flags the following sequence of statements as an error:

```
int vacationDays;
System.out.println(vacationDays); // ERROR--variable not initialized
```

You assign to a previously declared variable by using the variable name on the left, an equal sign (=), and then some Java expression that has an appropriate value on the right.

```
int vacationDays;
vacationDays = 12;
```

You can both declare and initialize a variable on the same line. For example:

```
int vacationDays = 12;
```

Finally, in Java you can put declarations anywhere in your code. For example, the following is valid code in Java:

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // ok to declare a variable here
```

In Java, it is considered good style to declare variables as closely as possible to the point where they are first used.

C++ NOTE: C and C++ distinguish between the *declaration* and *definition* of variables. For example,

```
int i = 10;
```

is a definition, whereas

```
extern int i;
```

is a declaration. In Java, no declarations are separate from definitions.

Constants

In Java, you use the keyword final to denote a constant. For example:

```
public class Constants
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      double paperWidth = 8.5;
      double paperHeight = 11;
      System.out.println("Paper size in centimeters: "
         + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
   }
}
```

The keyword final indicates that you can assign to the variable once, and then its value is set once and for all. It is customary to name constants in all uppercase.

It is probably more common in Java to want a constant that is available to multiple methods inside a single class. These are usually called *class constants.* You set up a class constant with the keywords static final. Here is an example of using a class constant:

```
public class Constants2
{
   public static void main(String[] args)
   {
      double paperWidth = 8.5;
      double paperHeight = 11;
      System.out.println("Paper size in centimeters: "
         + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }

   public static final double CM_PER_INCH = 2.54;
}
```

Note that the definition of the class constant appears *outside* the main method. Thus, the constant can also be used in other methods of the same class. Furthermore, if (as in our example) the constant is declared public, methods of other classes can also use the constant—in our example, as Constants2.CM_PER_INCH.

C++ NOTE: const is a reserved Java keyword, but it is not currently used for anything. You must use final for a constant.

Operators

The usual arithmetic operators + – * / are used in Java for addition, subtraction, multiplication, and division. The / operator denotes integer division if both arguments are integers, and floating-point division otherwise. Integer remainder (sometimes called *modulus*) is denoted by %. For example, 15 / 2 is 7, 15 % 2 is 1, and 15.0 / 2 is 7.5.

Note that integer division by 0 raises an exception, whereas floating-point division by 0 yields an infinite or NaN result.

There is a convenient shortcut for using binary arithmetic operators in an assignment. For example,

```
x += 4;
```

is equivalent to

```
x = x + 4;
```

(In general, place the operator to the left of the = sign, such as *= or %=.)

NOTE: One of the stated goals of the Java programming language is portability. A computation should yield the same results no matter which virtual machine executes it. For arithmetic computations with floating-point numbers, it is surprisingly difficult to achieve this portability. The `double` type uses 64 bits to store a numeric value, but some processors use 80-bit floating-point registers. These registers yield added precision in intermediate steps of a computation. For example, consider the following computation:

```
double w = x * y / z;
```

Many Intel processors compute `x * y` and leave the result in an 80-bit register, then divide by `z`, and finally truncate the result back to 64 bits. That can yield a more accurate result, and it can avoid exponent overflow. But the result may be *different* than a computation that uses 64 bits throughout. For that reason, the initial specification of the Java virtual machine mandated that all intermediate computations must be truncated. The numeric community hated it. Not only can the truncated computations cause overflow, they are actually *slower* than the more precise computations because the truncation operations take time. For that reason, the Java programming language was updated to recognize the conflicting demands for optimum performance and perfect reproducibility. By default, virtual machine designers are now permitted to use extended precision for intermediate computations. However, methods tagged with the `strictfp` keyword must use strict floating-point operations that yield reproducible results. For example, you can tag `main` as

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

Then all instructions inside the `main` method use strict floating-point computations. If you tag a class as `strictfp`, then all of its methods use strict floating-point computations.

The gory details are very much tied to the behavior of the Intel processors. In default mode, intermediate results are allowed to use an extended exponent, but not an extended mantissa. (The Intel chips support truncation of the mantissa without loss of performance.) Therefore, the only difference between default and strict mode is that strict computations may overflow when default computations don't.

If your eyes glazed over when reading this note, don't worry. Floating-point overflow isn't a problem that one encounters for most common programs. We don't use the `strictfp` keyword in this book.

Increment and Decrement Operators

Programmers, of course, know that one of the most common operations with a numeric variable is to add or subtract 1. Java, following in the footsteps of C and C++, has both increment and decrement operators: `n++` adds 1 to the current value of the variable `n`, and `n--` subtracts 1 from it. For example, the code

```
int n = 12;
n++;
```

changes n to 13. Because these operators change the value of a variable, they cannot be applied to numbers themselves. For example, 4++ is not a legal statement.

There are actually two forms of these operators; you have seen the "postfix" form of the operator that is placed after the operand. There is also a prefix form, ++n. Both change the value of the variable by 1. The difference between the two only appears when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

We recommend against using ++ inside other expressions because this often leads to confusing code and annoying bugs.

(Of course, while it is true that the ++ operator gives the C++ language its name, it also led to the first joke about the language. C++ haters point out that even the name of the language contains a bug: "After all, it should really be called ++C, because we only want to use a language after it has been improved.")

Relational and `boolean` *Operators*

Java has the full complement of relational operators. To test for equality you use a double equal sign, ==. For example, the value of

```
3 == 7
```

is false.

Use a != for inequality. For example, the value of

```
3 != 7
```

is true.

Finally, you have the usual < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators.

Java, following C++, uses && for the logical "and" operator and || for the logical "or" operator. As you can easily remember from the != operator, the exclamation point ! is the logical negation operator. The && and || operators are evaluated in "short circuit" fashion. The second argument is not evaluated if the first argument already determines the value. If you combine two expressions with the && operator,

expression$_1$ && *expression*$_2$

and the truth value of the first expression has been determined to be false, then it is impossible for the result to be true. Thus, the value for the second expression is *not* calculated. This behavior can be exploited to avoid errors. For example, in the expression

```
x != 0 && 1 / x > x + y // no division by 0
```

the second part is never evaluated if x equals zero. Thus, 1 / x is not computed if x is zero, and no divide-by-zero error can occur.

Similarly, the value of *expression*$_1$ || *expression*$_2$ is automatically true if the first expression is true, without evaluation of the second expression.

Finally, Java supports the ternary ?: operator that is occasionally useful. The expression

condition ? $expression_1$: $expression_2$

evaluates to the first expression if the condition is `true`, to the second expression otherwise. For example,

```
x < y ? x : y
```

gives the smaller of `x` and `y`.

Bitwise Operators

When working with any of the integer types, you have operators that can work directly with the bits that make up the integers. This means that you can use masking techniques to get at individual bits in a number. The bitwise operators are

`&` ("and") `|` ("or") `^` ("xor") `~` ("not")

These operators work on bit patterns. For example, if `n` is an integer variable, then

```
int fourthBitFromRight = (n & 8) / 8;
```

gives you a 1 if the fourth bit from the right in the binary representation of `n` is 1, and 0 if not. Using `&` with the appropriate power of 2 lets you mask out all but a single bit.

NOTE: When applied to `boolean` values, the `&` and `|` operators yield a `boolean` value. These operators are similar to the `&&` and `||` operators, except that the `&` and `|` operators are not evaluated in "short circuit" fashion. That is, both arguments are first evaluated before the result is computed.

There are also `>>` and `<<` operators, which shift a bit pattern to the right or left. These operators are often convenient when you need to build up bit patterns to do bit masking:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

Finally, a `>>>` operator fills the top bits with zero, whereas `>>` extends the sign bit into the top bits. There is no `<<<` operator.

CAUTION: The right-hand side argument of the shift operators is reduced modulo 32 (unless the left-hand side is a `long`, in which case the right-hand side is reduced modulo 64). For example, the value of `1 << 35` is the same as `1 << 3` or 8.

C++ NOTE: In C/C++, there is no guarantee as to whether >> performs an arithmetic shift (extending the sign bit) or a logical shift (filling in with zeroes). Implementors are free to choose whatever is more efficient. That means the C/C++ >> operator is really only defined for non-negative numbers. Java removes that ambiguity.

Mathematical Functions and Constants

The `Math` class contains an assortment of mathematical functions that you may occasionally need, depending on the kind of programming that you do.

To take the square root of a number, you use the `sqrt` method:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

NOTE: There is a subtle difference between the `println` method and the `sqrt` method. The `println` method operates on an object, `System.out`, defined in the `System` class. But the `sqrt` method in the `Math` class does not operate on any object. Such a method is called a *static* method. You can learn more about static methods in Chapter 4.

The Java programming language has no operator for raising a quantity to a power: You must use the `pow` method in the `Math` class. The statement

```
double y = Math.pow(x, a);
```

sets `y` to be `x` raised to the power `a` (x^a). The `pow` method has parameters that are both of type `double`, and it returns a `double` as well.

The `Math` class supplies the usual trigonometric functions

```
Math.sin
Math.cos
Math.tan
Math.atan
Math.atan2
```

and the exponential function and its inverse, the natural log:

```
Math.exp
Math.log
```

Finally, two constants denote the closest possible approximations to the mathematical constants π and *e*:

```
Math.PI
Math.E
```

TIP: Starting with Java SE 5.0, you can avoid the `Math` prefix for the mathematical methods and constants by adding the following line to the top of your source file:

```
import static java.lang.Math.*;
```

For example:

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

We discuss static imports in Chapter 4.

NOTE: The functions in the `Math` class use the routines in the computer's floating-point unit for fastest performance. If completely predictable results are more important than fast performance, use the `StrictMath` class instead. It implements the algorithms from the "Freely Distributable Math Library" `fdlibm`, guaranteeing identical results on all platforms. See `http://www.netlib.org/fdlibm/index.html` for the source of these algorithms. (Whenever `fdlibm` provides more than one definition for a function, the `StrictMath` class follows the IEEE 754 version whose name starts with an "e".)

Conversions between Numeric Types

It is often necessary to convert from one numeric type to another. Figure 3–1 shows the legal conversions.

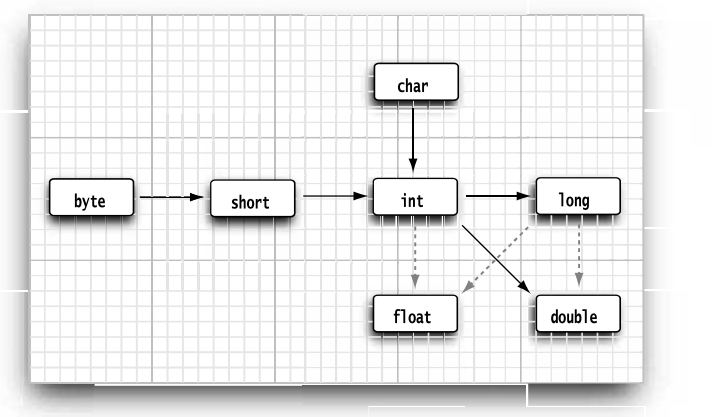

Figure 3–1 Legal conversions between numeric types

The six solid arrows in Figure 3–1 denote conversions without information loss. The three dotted arrows denote conversions that may lose precision. For example, a large integer such as 123456789 has more digits than the `float` type can represent. When the integer is converted to a `float`, the resulting value has the correct magnitude but it loses some precision.

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

When two values with a binary operator (such as `n + f` where `n` is an integer and `f` is a floating-point value) are combined, both operands are converted to a common type before the operation is carried out.

- If either of the operands is of type `double`, the other one will be converted to a `double`.
- Otherwise, if either of the operands is of type `float`, the other one will be converted to a `float`.
- Otherwise, if either of the operands is of type `long`, the other one will be converted to a `long`.
- Otherwise, both operands will be converted to an `int`.

Casts

In the preceding section, you saw that `int` values are automatically converted to `double` values when necessary. On the other hand, there are obviously times when you want to consider a `double` as an integer. Numeric conversions are possible in Java, but of course information may be lost. Conversions in which loss of information is possible are done by means of *casts*. The syntax for casting is to give the target type in parentheses, followed by the variable name. For example:

```
double x = 9.997;
int nx = (int) x;
```

Then, the variable nx has the value 9 because casting a floating-point value to an integer discards the fractional part.

If you want to *round* a floating-point number to the *nearest* integer (which is the more useful operation in most cases), use the Math.round method:

```
double x = 9.997;
int nx = (int) Math.round(x);
```

Now the variable nx has the value 10. You still need to use the cast (int) when you call round. The reason is that the return value of the round method is a long, and a long can only be assigned to an int with an explicit cast because there is the possibility of information loss.

CAUTION: If you try to cast a number of one type to another that is out of the range for the target type, the result will be a truncated number that has a different value. For example, (byte) 300 is actually 44.

C++ NOTE: You cannot cast between boolean values and any numeric type. This convention prevents common errors. In the rare case that you want to convert a boolean value to a number, you can use a conditional expression such as b ? 1 : 0.

Parentheses and Operator Hierarchy

Table 3–4 on the following page shows the precedence of operators. If no parentheses are used, operations are performed in the hierarchical order indicated. Operators on the same level are processed from left to right, except for those that are right associative, as indicated in the table. For example, because && has a higher precedence than ||, the expression

```
a && b || c
```

means

```
(a && b) || c
```

Because += associates right to left, the expression

```
a += b += c
```

means

```
a += (b += c)
```

That is, the value of b += c (which is the value of b after the addition) is added to a.

C++ NOTE: Unlike C or C++, Java does not have a comma operator. However, you can use a *comma-separated list of expressions* in the first and third slot of a for statement.

Table 3–4 Operator Precedence

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

Enumerated Types

Sometimes, a variable should only hold a restricted set of values. For example, you may sell clothes or pizza in four sizes: small, medium, large, and extra large. Of course, you could encode these sizes as integers 1, 2, 3, 4, or characters `S`, `M`, `L`, and `X`. But that is an error-prone setup. It is too easy for a variable to hold a wrong value (such as 0 or `m`).

Starting with Java SE 5.0, you can define your own *enumerated type* whenever such a situation arises. An enumerated type has a finite number of named values. For example:

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

Now you can declare variables of this type:

```
Size s = Size.MEDIUM;
```

A variable of type `Size` can hold only one of the values listed in the type declaration or the special value `null` that indicates that the variable is not set to any value at all.

We discuss enumerated types in greater detail in Chapter 5.

Strings

Conceptually, Java strings are sequences of Unicode characters. For example, the string `"Java\u2122"` consists of the five Unicode characters J, a, v, a, and ™. Java does not have a built-in string type. Instead, the standard Java library contains a predefined class called, naturally enough, `String`. Each quoted string is an instance of the `String` class:

```
String e = ""; // an empty string
String greeting = "Hello";
```

Substrings

You extract a substring from a larger string with the `substring` method of the `String` class. For example,

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

creates a string consisting of the characters `"Hel"`.

The second parameter of `substring` is the first position that you *do not* want to copy. In our case, we want to copy positions 0, 1, and 2 (from position 0 to position 2 inclusive). As `substring` counts it, this means from position 0 inclusive to position 3 *exclusive*.

There is one advantage to the way `substring` works: Computing the length of the substring is easy. The string `s.substring(a, b)` always has length `b - a`. For example, the substring `"Hel"` has length 3 – 0 = 3.

Concatenation

Java, like most programming languages, allows you to use the `+` sign to join (concatenate) two strings.

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

The preceding code sets the variable `message` to the string `"Expletivedeleted"`. (Note the lack of a space between the words: The `+` sign joins two strings in the order received, *exactly* as they are given.)

When you concatenate a string with a value that is not a string, the latter is converted to a string. (As you will see in Chapter 5, every Java object can be converted to a string.) For example,

```
int age = 13;
String rating = "PG" + age;
```

sets `rating` to the string `"PG13"`.

This feature is commonly used in output statements. For example,

```
System.out.println("The answer is " + answer);
```

is perfectly acceptable and will print what one would want (and with the correct spacing because of the space after the word `is`).

Strings Are Immutable

The `String` class gives no methods that let you *change* a character in an existing string. If you want to turn `greeting` into `"Help!"`, you cannot directly change the last positions of `greeting` into `'p'` and `'!'`. If you are a C programmer, this will make you feel pretty helpless. How are you going to modify the string? In Java, it is quite easy: Concatenate the substring that you want to keep with the characters that you want to replace.

```
greeting = greeting.substring(0, 3) + "p!";
```

This declaration changes the current value of the `greeting` variable to `"Help!"`.

Because you cannot change the individual characters in a Java string, the documentation refers to the objects of the `String` class as being *immutable.* Just as the number 3 is always 3, the string `"Hello"` will always contain the code unit sequence describing the characters H, e, l, l, o. You cannot change these values. You can, as you just saw however, change the contents of the string *variable* `greeting` and make it refer to a different string, just as you can make a numeric variable currently holding the value 3 hold the value 4.

Isn't that a lot less efficient? It would seem simpler to change the code units than to build up a whole new string from scratch. Well, yes and no. Indeed, it isn't efficient to generate a new string that holds the concatenation of `"Hel"` and `"p!"`. But immutable strings have one great advantage: the compiler can arrange that strings are *shared.*

To understand how this works, think of the various strings as sitting in a common pool. String variables then point to locations in the pool. If you copy a string variable, both the original and the copy share the same characters.

Overall, the designers of Java decided that the efficiency of sharing outweighs the inefficiency of string editing by extracting substrings and concatenating. Look at your own programs; we suspect that most of the time, you don't change strings—you just compare them. (There is one common exception—assembling strings from individual characters or shorter strings that come from the keyboard or a file. For these situations, Java provides a separate class that we describe in the section "Building Strings" on page 62.)

C++ NOTE: C programmers generally are bewildered when they see Java strings for the first time because they think of strings as arrays of characters:

```
char greeting[] = "Hello";
```

That is the wrong analogy: A Java string is roughly analogous to a `char*` pointer,

```
char* greeting = "Hello";
```

When you replace `greeting` with another string, the Java code does roughly the following:

```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "p!", 3);
greeting = temp;
```

Sure, now `greeting` points to the string `"Help!"`. And even the most hardened C programmer must admit that the Java syntax is more pleasant than a sequence of `strncpy` calls. But what if we make another assignment to `greeting`?

```
greeting = "Howdy";
```

Don't we have a memory leak? After all, the original string was allocated on the heap. Fortunately, Java does automatic garbage collection. If a block of memory is no longer needed, it will eventually be recycled.

If you are a C++ programmer and use the `string` class defined by ANSI C++, you will be much more comfortable with the Java `String` type. C++ `string` objects also perform automatic allocation and deallocation of memory. The memory management is performed explicitly by constructors, assignment operators, and destructors. However, C++ strings are mutable—you can modify individual characters in a string.

Testing Strings for Equality

To test whether two strings are equal, use the `equals` method. The expression

```
s.equals(t)
```

returns `true` if the strings `s` and `t` are equal, `false` otherwise. Note that `s` and `t` can be string variables or string constants. For example, the expression

```
"Hello".equals(greeting)
```

is perfectly legal. To test whether two strings are identical except for the upper/lowercase letter distinction, use the `equalsIgnoreCase` method.

```
"Hello".equalsIgnoreCase("hello")
```

Do *not* use the == operator to test whether two strings are equal! It only determines whether or not the strings are stored in the same location. Sure, if strings are in the same location, they must be equal. But it is entirely possible to store multiple copies of identical strings in different places.

```
String greeting = "Hello"; //initialize greeting to a string
if (greeting == "Hello") . . .
   // probably true
if (greeting.substring(0, 3) == "Hel") . . .
   // probably false
```

If the virtual machine would always arrange for equal strings to be shared, then you could use the == operator for testing equality. But only string *constants* are shared, not strings that are the result of operations like + or `substring`. Therefore, *never* use == to compare strings lest you end up with a program with the worst kind of bug—an intermittent one that seems to occur randomly.

C++ NOTE: If you are used to the C++ `string` class, you have to be particularly careful about equality testing. The C++ `string` class does overload the == operator to test for equality of the string contents. It is perhaps unfortunate that Java goes out of its way to give strings the same "look and feel" as numeric values but then makes strings behave like pointers for equality testing. The language designers could have redefined == for strings, just as they made a special arrangement for +. Oh well, every language has its share of inconsistencies.

C programmers never use == to compare strings but use `strcmp` instead. The Java method `compareTo` is the exact analog to `strcmp`. You can use

```
if (greeting.compareTo("Hello") == 0) . . .
```

but it seems clearer to use `equals` instead.

Code Points and Code Units

Java strings are implemented as sequences of `char` values. As we discussed in the section "The char Type" on page 42, the `char` data type is a code unit for representing Unicode code points in the UTF-16 encoding. The most commonly used Unicode characters can be represented with a single code unit. The supplementary characters require a pair of code units.

The `length` method yields the number of code units required for a given string in the UTF-16 encoding. For example:

```
String greeting = "Hello";
int n = greeting.length(); // is 5.
```

To get the true length, that is, the number of code points, call

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

The call `s.charAt(n)` returns the code unit at position `n`, where `n` is between 0 and `s.length() - 1`. For example:

```
char first = greeting.charAt(0); // first is 'H'
char last = greeting.charAt(4); // last is 'o'
```

To get at the ith code point, use the statements

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```

NOTE: Java counts the code units in strings in a peculiar fashion: the first code unit in a string has position 0. This convention originated in C, where there was a technical reason for counting positions starting at 0. That reason has long gone away and only the nuisance remains. However, so many programmers are used to this convention that the Java designers decided to keep it.

Why are we making a fuss about code units? Consider the sentence

```
𝕫 is the set of integers
```

The 𝕫 character requires two code units in the UTF-16 encoding. Calling

```
char ch = sentence.charAt(1)
```

doesn't return a space but the second code unit of 𝕫. To avoid this problem, you should not use the `char` type. It is too low-level.

If your code traverses a string, and you want to look at each code point in turn, use these statements:

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

Fortunately, the `codePointAt` method can tell whether a code unit is the first or second half of a supplementary character, and it returns the right result either way. That is, you can move backwards with the following statements:

```
i--;
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i--;
```

The `String` *API*

The `String` class in Java contains more than 50 methods. A surprisingly large number of them are sufficiently useful so that we can imagine using them frequently. The following API note summarizes the ones we found most useful.

NOTE: You will find these API notes throughout the book to help you understand the Java Application Programming Interface (API). Each API note starts with the name of a class such as `java.lang.String`—the significance of the so-called *package* name `java.lang` is explained in Chapter 4. The class name is followed by the names, explanations, and parameter descriptions of one or more methods.

We typically do not list all methods of a particular class but instead select those that are most commonly used, and describe them in a concise form. For a full listing, consult the online documentation (see "Reading the On-Line API Documentation" on page 59).

We also list the version number in which a particular class was introduced. If a method has been added later, it has a separate version number.

`java.lang.String` 1.0

- `char charAt(int index)`
  returns the code unit at the specified location. You probably don't want to call this method unless you are interested in low-level code units.
- `int codePointAt(int index)` **5.0**
  returns the code point that starts or ends at the specified location.
- `int offsetByCodePoints(int startIndex, int cpCount)` **5.0**
  returns the index of the code point that is `cpCount` code points away from the code point at `startIndex`.
- `int compareTo(String other)`
  returns a negative value if the string comes before `other` in dictionary order, a positive value if the string comes after `other` in dictionary order, or 0 if the strings are equal.
- `boolean endsWith(String suffix)`
  returns `true` if the string ends with `suffix`.
- `boolean equals(Object other)`
  returns `true` if the string equals `other`.
- `boolean equalsIgnoreCase(String other)`
  returns `true` if the string equals `other`, except for upper/lowercase distinction.
- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`
  returns the start of the first substring equal to the string `str` or the code point `cp`, starting at index 0 or at `fromIndex`, or –1 if `str` does not occur in this string.
- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`
- `int lastindexOf(int cp)`
- `int lastindexOf(int cp, int fromIndex)`
  returns the start of the last substring equal to the string `str` or the code point `cp`, starting at the end of the string or at `fromIndex`.

- `int length()`
  returns the length of the string.
- `int codePointCount(int startIndex, int endIndex)` **5.0**
  returns the number of code points between `startIndex` and `endIndex - 1`. Unpaired surrogates are counted as code points.
- `String replace(CharSequence oldString, CharSequence newString)`
  returns a new string that is obtained by replacing all substrings matching `oldString` in the string with the string `newString`. You can supply `String` or `StringBuilder` objects for the `CharSequence` parameters.
- `boolean startsWith(String prefix)`
  returns `true` if the string begins with `prefix`.
- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`
  returns a new string consisting of all code units from `beginIndex` until the end of the string or until `endIndex - 1`.
- `String toLowerCase()`
  returns a new string containing all characters in the original string, with uppercase characters converted to lowercase.
- `String toUpperCase()`
  returns a new string containing all characters in the original string, with lowercase characters converted to uppercase.
- `String trim()`
  returns a new string by eliminating all leading and trailing spaces in the original string.

Reading the On-Line API Documentation

As you just saw, the `String` class has lots of methods. Furthermore, there are thousands of classes in the standard libraries, with many more methods. It is plainly impossible to remember all useful classes and methods. Therefore, it is essential that you become familiar with the on-line API documentation that lets you look up all classes and methods in the standard library. The API documentation is part of the JDK. It is in HTML format. Point your web browser to the `docs/api/index.html` subdirectory of your JDK installation. You will see a screen like that in Figure 3–2.

The screen is organized into three frames. A small frame on the top left shows all available packages. Below it, a larger frame lists all classes. Click on any class name, and the API documentation for the class is displayed in the large frame to the right (see Figure 3–3). For example, to get more information on the methods of the `String` class, scroll the second frame until you see the `String` link, then click on it.

Then scroll the frame on the right until you reach a summary of all methods, sorted in alphabetical order (see Figure 3–4). Click on any method name for a detailed description of that method (see Figure 3–5). For example, if you click on the `compareToIgnoreCase` link, you get the description of the `compareToIgnoreCase` method.

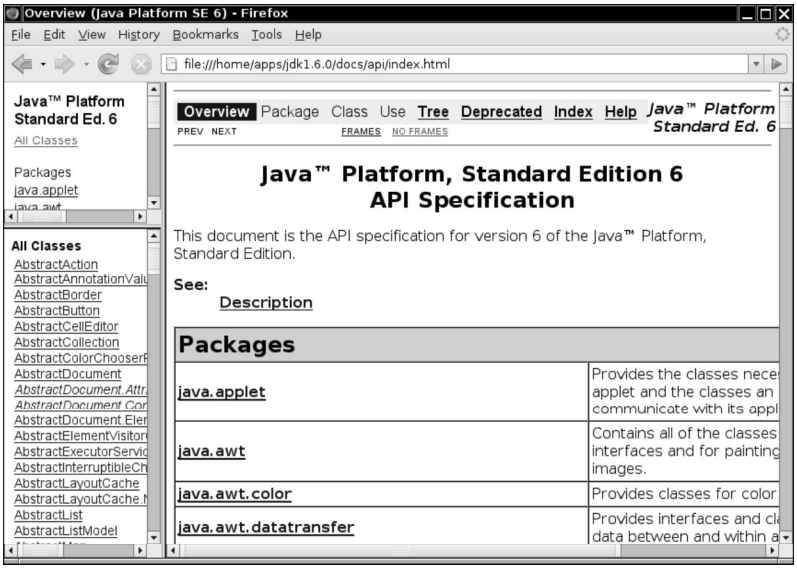

Figure 3–2 The three panes of the API documentation

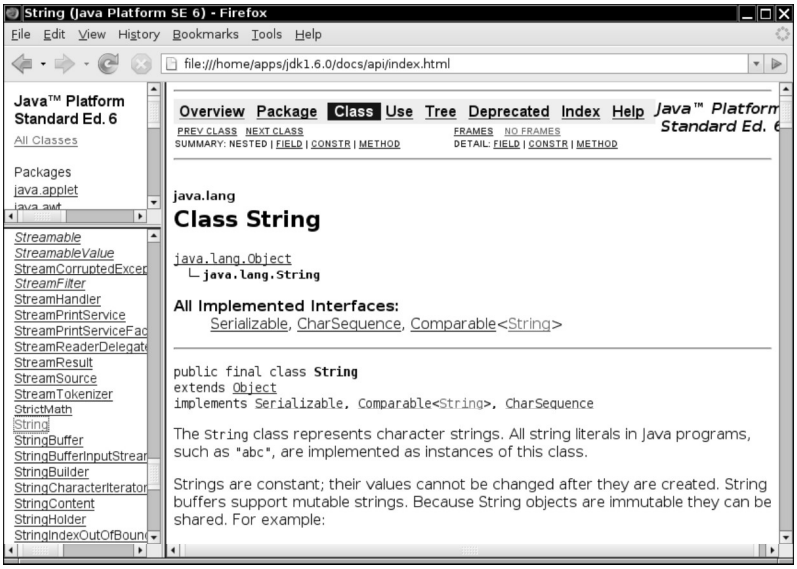

Figure 3–3 Class description for the `String` class

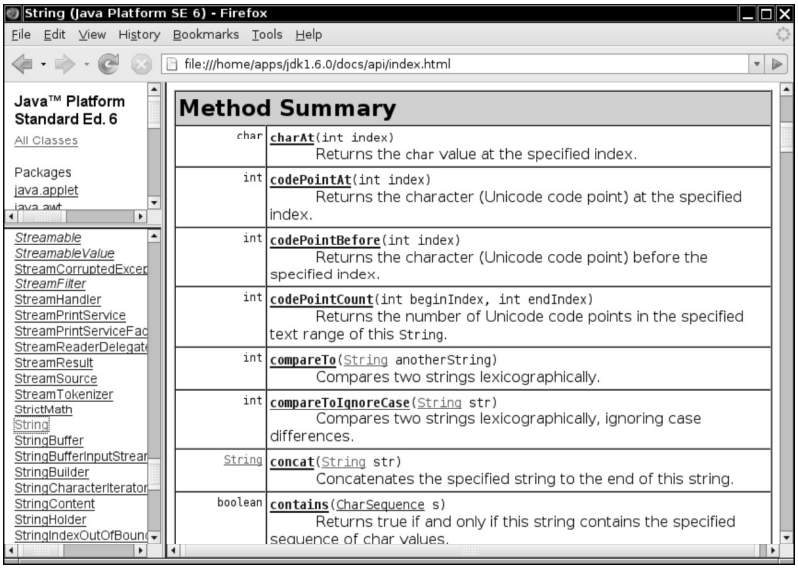

Figure 3–4 Method summary of the `String` class

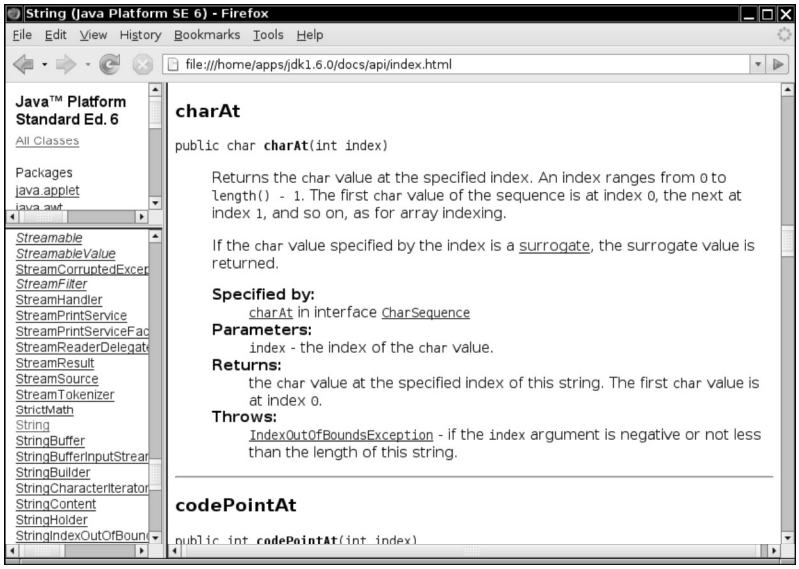

Figure 3–5 Detailed description of a `String` method

TIP: Bookmark the docs/api/index.html page in your browser right now.

Building Strings

Occasionally, you need to build up strings from shorter strings, such as keystrokes or words from a file. It would be inefficient to use string concatenation for this purpose. Every time you concatenate strings, a new String object is constructed. This is time consuming and it wastes memory. Using the StringBuilder class avoids this problem.

Follow these steps if you need to build a string from many small pieces. First, construct an empty string builder:

```
StringBuilder builder = new StringBuilder();
```

(We discuss constructors and the new operator in detail in Chapter 4.)

Each time you need to add another part, call the append method.

```
builder.append(ch); // appends a single character
builder.append(str); // appends a string
```

When you are done building the string, call the toString method. You will get a String object with the character sequence contained in the builder.

```
String completedString = builder.toString();
```

NOTE: The StringBuilder class was introduced in JDK 5.0. Its predecessor, StringBuffer, is slightly less efficient, but it allows multiple threads to add or remove characters. If all string editing happens in a single thread (which is usually the case), you should use StringBuilder instead. The APIs of both classes are identical.

The following API notes contain the most important methods for the StringBuilder class.

java.lang.StringBuilder 5.0

- StringBuilder()
  constructs an empty string builder.
- int length()
  returns the number of code units of the builder or buffer.
- StringBuilder append(String str)
  appends a string and returns this.
- StringBuilder append(char c)
  appends a code unit and returns this.
- StringBuilder appendCodePoint(int cp)
  appends a code point, converting it into one or two code units, and returns this.
- void setCharAt(int i, char c)
  sets the ith code unit to c.
- StringBuilder insert(int offset, String str)
  inserts a string at position offset and returns this.

- `StringBuilder insert(int offset, char c)`
  inserts a code unit at position `offset` and returns `this`.
- `StringBuilder delete(int startIndex, int endIndex)`
  deletes the code units with offsets `startIndex` to `endIndex - 1` and returns `this`.
- `String toString()`
  returns a string with the same data as the builder or buffer contents.

Input and Output

To make our example programs more interesting, we want to accept input and properly format the program output. Of course, modern programs use a GUI for collecting user input. However, programming such an interface requires more tools and techniques than we have at our disposal at this time. Because the first order of business is to become more familiar with the Java programming language, we make do with the humble console for input and output for now. GUI programming is covered in Chapters 7 through 9.

Reading Input

You saw that it is easy to print output to the "standard output stream" (that is, the console window) just by calling `System.out.println`. Reading from the "standard input stream" `System.in` isn't quite as simple. To read console input, you first construct a `Scanner` that is attached to `System.in`:

```
Scanner in = new Scanner(System.in);
```

(We discuss constructors and the `new` operator in detail in Chapter 4.)

Now you use the various methods of the `Scanner` class to read input. For example, the `nextLine` method reads a line of input.

```
System.out.print("What is your name? ");
String name = in.nextLine();
```

Here, we use the `nextLine` method because the input might contain spaces. To read a single word (delimited by whitespace), call

```
String firstName = in.next();
```

To read an integer, use the `nextInt` method.

```
System.out.print("How old are you? ");
int age = in.nextInt();
```

Similarly, the `nextDouble` method reads the next floating-point number.

The program in Listing 3–2 asks for the user's name and age and then prints a message like

```
Hello, Cay. Next year, you'll be 46
```

Finally, note the line

```
import java.util.*;
```

at the beginning of the program. The `Scanner` class is defined in the `java.util` package. Whenever you use a class that is not defined in the basic `java.lang` package, you need to use an `import` directive. We look at packages and `import` directives in more detail in Chapter 4.

Listing 3–2 InputTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates console input.
5.  * @version 1.10 2004-02-10
6.  * @author Cay Horstmann
7.  */
8. public class InputTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Scanner in = new Scanner(System.in);
13.
14.       // get first input
15.       System.out.print("What is your name? ");
16.       String name = in.nextLine();
17.
18.       // get second input
19.       System.out.print("How old are you? ");
20.       int age = in.nextInt();
21.
22.       // display output on console
23.       System.out.println("Hello, " + name + ". Next year, you'll be " + (age + 1));
24.    }
25. }
```

NOTE: The Scanner class is not suitable for reading a password from a console since the input is plainly visible to anyone. Java SE 6 introduces a Console class specifically for this purpose. To read a password, use the following code:

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

For security reasons, the password is returned in an array of characters rather than a string. After you are done processing the password, you should immediately overwrite the array elements with a filler value. (Array processing is discussed later in this chapter.)

Input processing with a Console object is not as convenient as with a Scanner. You can only read a line of input at a time. There are no methods for reading individual words or numbers.

java.util.Scanner 5.0

- `Scanner(InputStream in)`
  constructs a Scanner object from the given input stream.
- `String nextLine()`
  reads the next line of input.

- `String next()`
  reads the next word of input (delimited by whitespace).
- `int nextInt()`
- `double nextDouble()`
  reads and converts the next character sequence that represents an integer or floating-point number.
- `boolean hasNext()`
  tests whether there is another word in the input.
- `boolean hasNextInt()`
- `boolean hasNextDouble()`
  tests whether the next character sequence represents an integer or floating-point number.

API `java.lang.System 1.0`

- `static Console console()` **6**
  returns a `Console` object for interacting with the user through a console window if such an interaction is possible, `null` otherwise. A `Console` object is available for any program that is launched in a console window. Otherwise, the availability is system-dependent.

API `java.io.Console 6`

- `static char[] readPassword(String prompt, Object... args)`
- `static String readLine(String prompt, Object... args)`
  displays the prompt and reads the user input until the end of the input line. The `args` parameters can be used to supply formatting arguments, as described in the next section.

Formatting Output

You can print a number `x` to the console with the statement `System.out.print(x)`. That command will print `x` with the maximum number of non-zero digits for that type. For example,

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

prints

```
3333.3333333333335
```

That is a problem if you want to display, for example, dollars and cents.

In early versions of Java, formatting numbers was a bit of a hassle. Fortunately, Java SE 5.0 brought back the venerable `printf` method from the C library. For example, the call

```
System.out.printf("%8.2f", x);
```

prints `x` with a *field width* of 8 characters and a *precision* of 2 characters. That is, the printout contains a leading space and the seven characters

```
3333.33
```

You can supply multiple parameters to `printf`. For example:

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

Each of the *format specifiers* that start with a `%` character is replaced with the corresponding argument. The *conversion character* that ends a format specifier indicates the type of the value to be formatted: `f` is a floating-point number, `s` a string, and `d` a decimal integer. Table 3–5 shows all conversion characters.

Table 3–5 Conversions for `printf`

Conversion Character	Type	Example
`d`	Decimal integer	`159`
`x`	Hexadecimal integer	`9f`
`o`	Octal integer	`237`
`f`	Fixed-point floating-point	`15.9`
`e`	Exponential floating-point	`1.59e+01`
`g`	General floating-point (the shorter of `e` and `f`)	—
`a`	Hexadecimal floating-point	`0x1.fccdp3`
`s`	String	`Hello`
`c`	Character	`H`
`b`	`boolean`	`true`
`h`	Hash code	`42628b2`
`t`*x*	Date and time	See Table 3–7
`%`	The percent symbol	`%`
`n`	The platform-dependent line separator	—

In addition, you can specify *flags* that control the appearance of the formatted output. Table 3–6 shows all flags. For example, the comma flag adds group separators. That is,

```
System.out.printf("%,.2f", 10000.0 / 3.0);
```

prints

```
3,333.33
```

You can use multiple flags, for example, `"%,(.2f"`, to use group separators and enclose negative numbers in parentheses.

NOTE: You can use the `s` conversion to format arbitrary objects. If an arbitrary object implements the `Formattable` interface, the object's `formatTo` method is invoked. Otherwise, the `toString` method is invoked to turn the object into a string. We discuss the `toString` method in Chapter 5 and interfaces in Chapter 6.

Table 3–6 Flags for `printf`

Flag	Purpose	Example
+	Prints sign for positive and negative numbers	+3333.33
space	Adds a space before positive numbers	| 3333.33|
0	Adds leading zeroes	003333.33
-	Left-justifies field	|3333.33 |
(	Encloses negative number in parentheses	(3333.33)
,	Adds group separators	3,333.33
# (for f format)	Always includes a decimal point	3,333.
# (for x or o format)	Adds 0x or 0 prefix	0xcafe
$	Specifies the index of the argument to be formatted; for example, %1$d %1$x prints the first argument in decimal and hexadecimal	159 9F
<	Formats the same value as the previous specification; for example, %d %<x prints the same number in decimal and hexadecimal	159 9F

You can use the static `String.format` method to create a formatted string without printing it:

```
String message = String.format("Hello, %s. Next year, you'll be %d", name, age);
```

Although we do not describe the `Date` type in detail until Chapter 4, we do, in the interest of completeness, briefly discuss the date and time formatting options of the `printf` method. You use a two-letter format, starting with `t` and ending in one of the letters of Table 3–7. For example,

```
System.out.printf("%tc", new Date());
```

prints the current date and time in the format

```
Mon Feb 09 18:05:19 PST 2004
```

Table 3–7 Date and Time Conversion Characters

Conversion Character	Type	Example
c	Complete date and time	Mon Feb 09 18:05:19 PST 2004
F	ISO 8601 date	2004-02-09
D	U.S. formatted date (month/day/year)	02/09/2004
T	24-hour time	18:05:19

Table 3–7 Date and Time Conversion Characters (continued)

Conversion Character	Type	Example
r	12-hour time	06:05:19 pm
R	24-hour time, no seconds	18:05
Y	Four-digit year (with leading zeroes)	2004
y	Last two digits of the year (with leading zeroes)	04
C	First two digits of the year (with leading zeroes)	20
B	Full month name	February
b or h	Abbreviated month name	Feb
m	Two-digit month (with leading zeroes)	02
d	Two-digit day (with leading zeroes)	09
e	Two-digit day (without leading zeroes)	9
A	Full weekday name	Monday
a	Abbreviated weekday name	Mon
j	Three-digit day of year (with leading zeroes), between 001 and 366	069
H	Two-digit hour (with leading zeroes), between 00 and 23	18
k	Two-digit hour (without leading zeroes), between 0 and 23	18
I	Two-digit hour (with leading zeroes), between 01 and 12	06
l	Two-digit hour (without leading zeroes), between 1 and 12	6
M	Two-digit minutes (with leading zeroes)	05
S	Two-digit seconds (with leading zeroes)	19
L	Three-digit milliseconds (with leading zeroes)	047
N	Nine-digit nanoseconds (with leading zeroes)	047000000
P	Uppercase morning or afternoon marker	PM
p	Lowercase morning or afternoon marker	pm
z	RFC 822 numeric offset from GMT	-0800
Z	Time zone	PST
s	Seconds since 1970-01-01 00:00:00 GMT	1078884319
Q	Milliseconds since 1970-01-01 00:00:00 GMT	1078884319047

As you can see in Table 3–7, some of the formats yield only a part of a given date, for example, just the day or just the month. It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the *index* of the argument to be formatted. The index must immediately follow the `%`, and it must be terminated by a `$`. For example,

```
System.out.printf("%1$s %2$tB %2$te, %2$tY", "Due date:", new Date());
```

prints

```
Due date: February 9, 2004
```

Alternatively, you can use the `<` flag. It indicates that the same argument as in the preceding format specification should be used again. That is, the statement

```
System.out.printf("%s %tB %<te, %<tY", "Due date:", new Date());
```

yields the same output as the preceding statement.

CAUTION: Argument index values start with 1, not with 0: `%1$...` formats the first argument. This avoids confusion with the `0` flag.

You have now seen all features of the `printf` method. Figure 3–6 shows a syntax diagram for format specifiers.

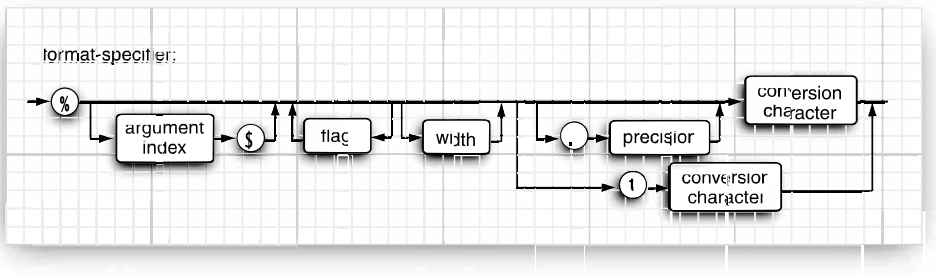

Figure 3–6 Format specifier syntax

NOTE: A number of the formatting rules are *locale specific*. For example, in Germany, the decimal separator is a period, not a comma, and Monday is formatted as Montag. You will see in Volume II how to control the international behavior of your applications.

File Input and Output

To read from a file, construct a `Scanner` object from a `File` object, like this:

```
Scanner in = new Scanner(new File("myfile.txt"));
```

If the file name contains backslashes, remember to escape each of them with an additional backslash: `"c:\\mydirectory\\myfile.txt"`.

Now you can read from the file, using any of the `Scanner` methods that we already described.

To write to a file, construct a `PrintWriter` object. In the constructor, simply supply the file name:

```
PrintWriter out = new PrintWriter("myfile.txt");
```

If the file does not exist, you can simply use the `print`, `println`, and `printf` commands as you did when printing to `System.out`.

CAUTION: You can construct a `Scanner` with a string parameter, but the scanner interprets the string as data, not a file name. For example, if you call

```
Scanner in = new Scanner("myfile.txt"); // ERROR?
```

then the scanner will see ten characters of data: `'m'`, `'y'`, `'f'`, and so on. That is probably not what was intended in this case.

NOTE: When you specify a relative file name, such as `"myfile.txt"`, `"mydirectory/myfile.txt"`, or `"../myfile.txt"`, the file is located relative to the directory in which the Java virtual machine was started. If you launched your program from a command shell, by executing

```
java MyProg
```

then the starting directory is the current directory of the command shell. However, if you use an integrated development environment, the starting directory is controlled by the IDE. You can find the directory location with this call:

```
String dir = System.getProperty("user.dir");
```

If you run into grief with locating files, consider using absolute path names such as `"c:\\mydirectory\\myfile.txt"` or `"/home/me/mydirectory/myfile.txt"`.

As you just saw, you can access files just as easily as you can use `System.in` and `System.out`. There is just one catch: If you construct a `Scanner` with a file that does not exist or a `PrintWriter` with a file name that cannot be created, an exception occurs. The Java compiler considers these exceptions to be more serious than a "divide by zero" exception, for example. In Chapter 11, you will learn various ways for handing exceptions. For now, you should simply tell the compiler that you are aware of the possibility of a "file not found" exception. You do this by tagging the `main` method with a `throws` clause, like this:

```
public static void main(String[] args) throws FileNotFoundException
{
   Scanner in = new Scanner(new File("myfile.txt"));
   . . .
}
```

You have now seen how to read and write files that contain textual data. For more advanced topics, such as dealing with different character encodings, processing binary data, reading directories, and writing zip files, please turn to Chapter 1 of Volume II.

NOTE: When you launch a program from a command shell, you can use the redirection syntax of your shell and attach any file to `System.in` and `System.out`:

```
java MyProg < myfile.txt > output.txt
```

Then, you need not worry about handling the `FileNotFoundException`.

java.util.Scanner 5.0

- `Scanner(File f)`
  constructs a `Scanner` that reads data from the given file.
- `Scanner(String data)`
  constructs a `Scanner` that reads data from the given string.

java.io.PrintWriter 1.1

- `PrintWriter(File f)`
  constructs a `PrintWriter` that writes data to the given file.
- `PrintWriter(String fileName)`
  constructs a `PrintWriter` that writes data to the file with the given file name.

java.io.File 1.0

- `File(String fileName)`
  constructs a `File` object that describes a file with the given name. Note that the file need not currently exist.

Control Flow

Java, like any programming language, supports both conditional statements and loops to determine control flow. We start with the conditional statements and then move on to loops. We end with the somewhat cumbersome `switch` statement that you can use when you have to test for many values of a single expression.

C++ NOTE: The Java control flow constructs are identical to those in C and C++, with a few exceptions. There is no `goto`, but there is a "labeled" version of `break` that you can use to break out of a nested loop (where you perhaps would have used a `goto` in C). Finally! Java SE 5.0 added a variant of the `for` loop that has no analog in C or C++. It is similar to the `foreach` loop in C#.

Block Scope

Before we get into the actual control structures, you need to know more about *blocks*.

A block or compound statement is any number of simple Java statements that are surrounded by a pair of braces. Blocks define the scope of your variables. Blocks can be *nested* inside another block. Here is a block that is nested inside the block of the `main` method.

```
public static void main(String[] args)
{
   int n;
   . . .
   {
      int k;
      . . .
   } // k is only defined up to here
}
```

However, you may not declare identically named variables in two nested blocks. For example, the following is an error and will not compile:

```
public static void main(String[] args)
{
   int n;
   . . .
   {
      int k;
      int n; // error--can't redefine n in inner block
      . . .
   }
}
```

C++ NOTE: In C++, it is possible to redefine a variable inside a nested block. The inner definition then shadows the outer one. This can be a source of programming errors; hence, Java does not allow it.

Conditional Statements

The conditional statement in Java has the form

```
if (condition) statement
```

The condition must be surrounded by parentheses.

In Java, as in most programming languages, you will often want to execute multiple statements when a single condition is true. In this case, you use a *block statement* that takes the form

```
{
   statement1
   statement2
   . . .
}
```

For example:

```
if (yourSales >= target)
{
   performance = "Satisfactory";
   bonus = 100;
}
```

In this code all the statements surrounded by the braces will be executed when `yourSales` is greater than or equal to `target` (see Figure 3–7).

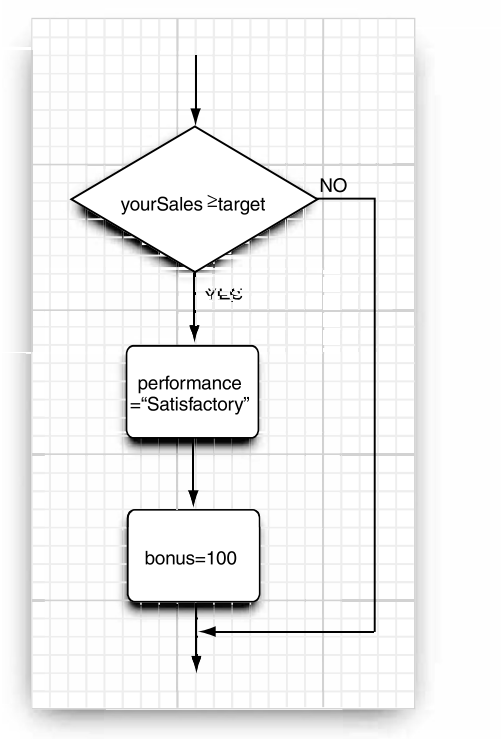

Figure 3–7 Flowchart for the `if` statement

NOTE: A block (sometimes called a *compound statement*) allows you to have more than one (simple) statement in any Java programming structure that might otherwise have a single (simple) statement.

The more general conditional in Java looks like this (see Figure 3–8):

`if` (*condition*) *statement*$_1$ `else` *statement*$_2$

For example:

```
if (yourSales >= target)
{
   performance = "Satisfactory";
   bonus = 100 + 0.01 * (yourSales - target);
}
else
{
   performance = "Unsatisfactory";
   bonus = 0;
}
```

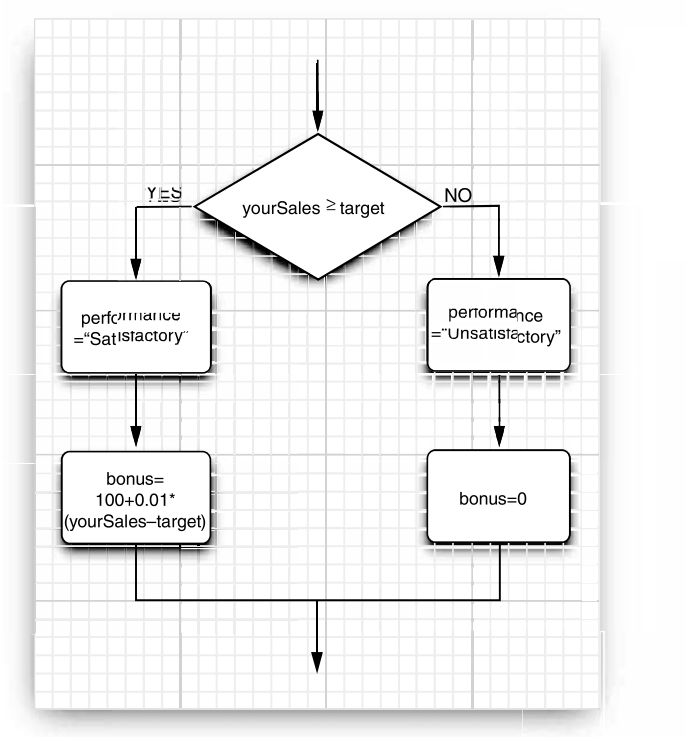

Figure 3–8 Flowchart for the `if/else` statement

The `else` part is always optional. An `else` groups with the closest `if`. Thus, in the statement

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

the `else` belongs to the second `if`. Of course, it is a good idea to use braces to clarify this code:

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

Repeated `if . . . else if . . .` alternatives are common (see Figure 3–9). For example:

```
if (yourSales >= 2 * target)
{
   performance = "Excellent";
   bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
   performance = "Fine";
   bonus = 500;
}
else if (yourSales >= target)
```

```
{
   performance = "Satisfactory";
   bonus = 100;
}
else
{
   System.out.println("You're fired");
}
```

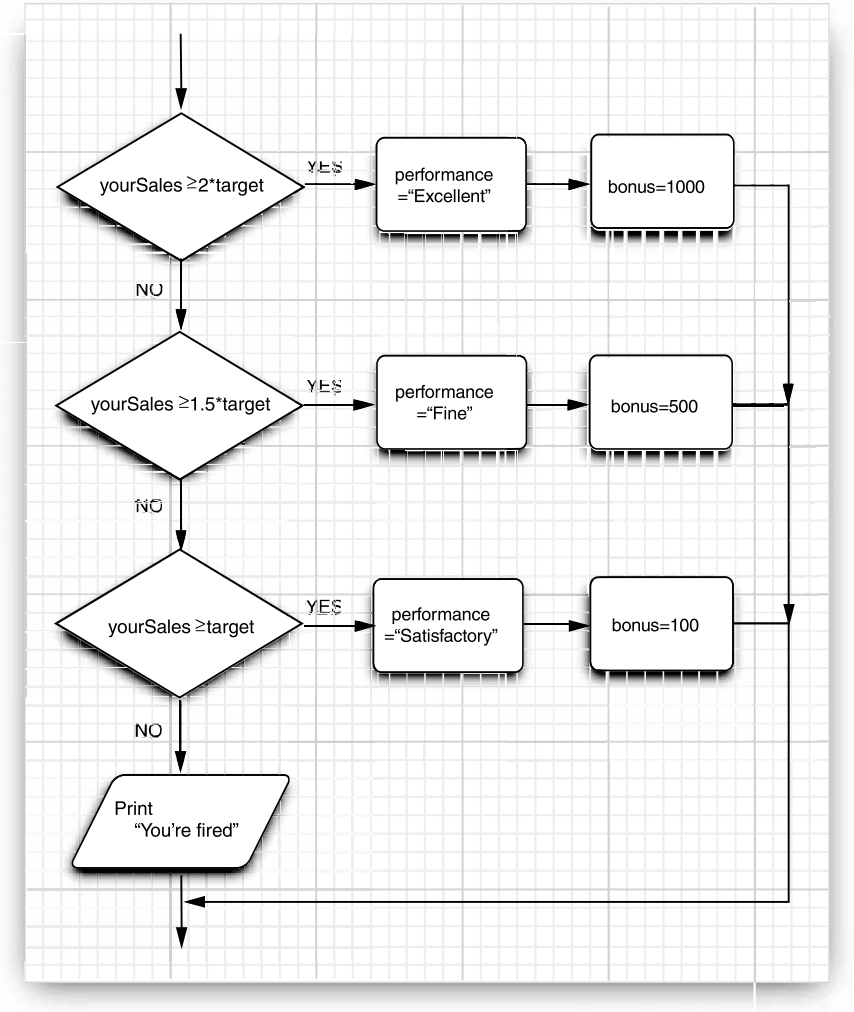

Figure 3–9 Flowchart for the `if/else if` (multiple branches)

Loops

The `while` loop executes a statement (which may be a block statement) while a condition is `true`. The general form is

`while` (*condition*) *statement*

The `while` loop will never execute if the condition is `false` at the outset (see Figure 3–10).

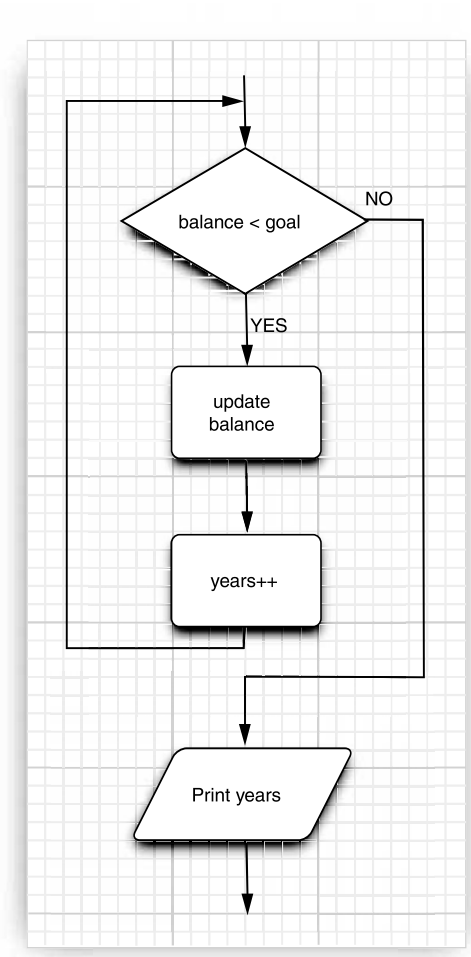

Figure 3–10 Flowchart for the `while` statement

The program in Listing 3–3 determines how long it will take to save a specific amount of money for your well-earned retirement, assuming that you deposit the same amount of money per year and that the money earns a specified interest rate.

In the example, we are incrementing a counter and updating the amount currently accumulated in the body of the loop until the total exceeds the targeted amount.

```
while (balance < goal)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   years++;
}
System.out.println(years + " years.");
```

(Don't rely on this program to plan for your retirement. We left out a few niceties such as inflation and your life expectancy.)

A `while` loop tests at the top. Therefore, the code in the block may never be executed. If you want to make sure a block is executed at least once, you will need to move the test to the bottom. You do that with the `do/while` loop. Its syntax looks like this:

```
do statement while (condition);
```

This loop executes the statement (which is typically a block) and only then tests the condition. It then repeats the statement and retests the condition, and so on. The code in Listing 3–4 computes the new balance in your retirement account and then asks if you are ready to retire:

```
do
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   year++;
   // print current balance
   . . .
   // ask if ready to retire and get input
   . . .
}
while (input.equals("N"));
```

As long as the user answers `"N"`, the loop is repeated (see Figure 3–11). This program is a good example of a loop that needs to be entered at least once, because the user needs to see the balance before deciding whether it is sufficient for retirement.

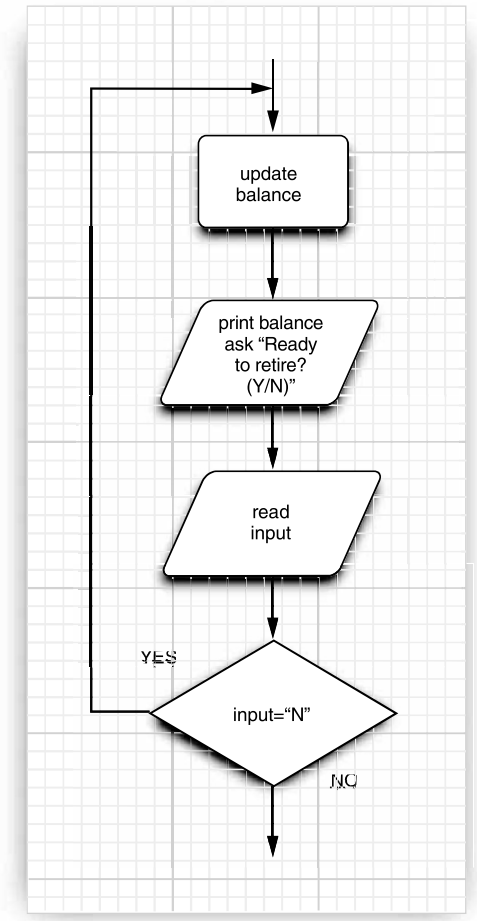

Figure 3–11 Flowchart for the `do/while` statement

Listing 3–3 Retirement.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates a <code>while</code> loop.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */
```

Listing 3–3 Retirement.java (continued)

```
public class Retirement
{
   public static void main(String[] args)
   {
      // read inputs
      Scanner in = new Scanner(System.in);

      System.out.print("How much money do you need to retire? ");
      double goal = in.nextDouble();

      System.out.print("How much money will you contribute every year? ");
      double payment = in.nextDouble();

      System.out.print("Interest rate in %: ");
      double interestRate = in.nextDouble();

      double balance = 0;
      int years = 0;

      // update account balance while goal isn't reached
      while (balance < goal)
      {
         // add this year's payment and interest
         balance += payment;
         double interest = balance * interestRate / 100;
         balance += interest;
         years++;
      }

      System.out.println("You can retire in " + years + " years.");
   }
}
```

Listing 3–4 Retirement2.java

```
import java.util.*;

/**
 * This program demonstrates a <code>do/while</code> loop.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class Retirement2
{
```

Listing 3–4 Retirement2.java (continued)

```
10.    public static void main(String[] args)
11.    {
12.       Scanner in = new Scanner(System.in);
13.
14.       System.out.print("How much money will you contribute every year? ");
15.       double payment = in.nextDouble();
16.
17.       System.out.print("Interest rate in %: ");
18.       double interestRate = in.nextDouble();
19.
20.       double balance = 0;
21.       int year = 0;
22.
23.       String input;
24.
25.       // update account balance while user isn't ready to retire
26.       do
27.       {
28.          // add this year's payment and interest
29.          balance += payment;
30.          double interest = balance * interestRate / 100;
31.          balance += interest;
32.
33.          year++;
34.
35.          // print current balance
36.          System.out.printf("After year %d, your balance is %,.2f%n", year, balance);
37.
38.          // ask if ready to retire and get input
39.          System.out.print("Ready to retire? (Y/N) ");
40.          input = in.next();
41.       }
42.       while (input.equals("N"));
43.    }
44. }
```

Determinate Loops

The for loop is a general construct to support iteration that is controlled by a counter or similar variable that is updated after every iteration. As Figure 3–12 shows, the following loop prints the numbers from 1 to 10 on the screen.

```
for (int i = 1; i <= 10; i++)
   System.out.println(i);
```

The first slot of the for statement usually holds the counter initialization. The second slot gives the condition that will be tested before each new pass through the loop, and the third slot explains how to update the counter.

Although Java, like C++, allows almost any expression in the various slots of a `for` loop, it is an unwritten rule of good taste that the three slots of a `for` statement should only initialize, test, and update the same counter variable. One can write very obscure loops by disregarding this rule.

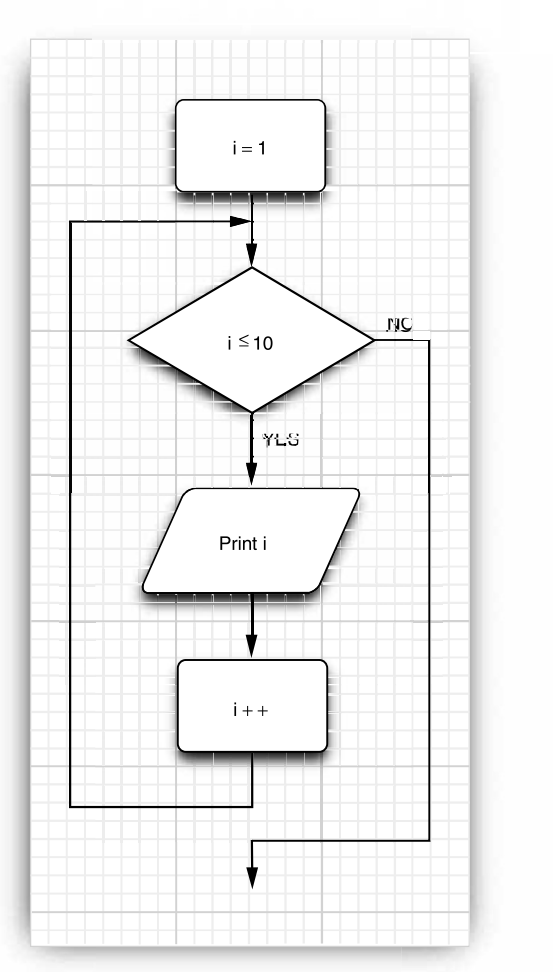

Figure 3–12 Flowchart for the `for` statement

Even within the bounds of good taste, much is possible. For example, you can have loops that count down:

```
for (int i = 10; i > 0; i--)
   System.out.println("Counting down . . . " + i);
System.out.println("Blastoff!");
```

CAUTION: Be careful about testing for equality of floating-point numbers in loops. A `for` loop that looks like

```
for (double x = 0; x != 10; x += 0.1) . . .
```

may never end. Because of roundoff errors, the final value may not be reached exactly. For example, in the loop above, `x` jumps from 9.99999999999998 to 10.09999999999998 because there is no exact binary representation for 0.1.

When you declare a variable in the first slot of the `for` statement, the scope of that variable extends until the end of the body of the `for` loop.

```
for (int i = 1; i <= 10; i++)
{
   . . .
}
// i no longer defined here
```

In particular, if you define a variable inside a `for` statement, you cannot use the value of that variable outside the loop. Therefore, if you wish to use the final value of a loop counter outside the `for` loop, be sure to declare it outside the loop header!

```
int i;
for (i = 1; i <= 10; i++)
{
   . . .
}
// i still defined here
```

On the other hand, you can define variables with the same name in separate `for` loops:

```
for (int i = 1; i <= 10; i++)
{
   . . .
}
. . .
for (int i = 11; i <= 20; i++) // ok to define another variable named i
{
   . . .
}
```

A `for` loop is merely a convenient shortcut for a `while` loop. For example,

```
for (int i = 10; i > 0; i--)
   System.out.println("Counting down . . . " + i);
```

can be rewritten as

```
int i = 10;
while (i > 0)
{
   System.out.println("Counting down . . . " + i);
   i--;
}
```

Listing 3–5 shows a typical example of a `for` loop.

The program computes the odds on winning a lottery. For example, if you must pick 6 numbers from the numbers 1 to 50 to win, then there are (50 × 49 × 48 × 47 × 46 × 45)/(1 × 2 × 3 × 4 × 5 × 6) possible outcomes, so your chance is 1 in 15,890,700. Good luck!

In general, if you pick k numbers out of n, there are

$$\frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{1 \times 2 \times 3 \times \ldots \times k}$$

possible outcomes. The following `for` loop computes this value:

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
   lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

NOTE: See “The ‘for each’ Loop” on page 91 for a description of the “generalized `for` loop” (also called “for each” loop) that was added to the Java language in Java SE 5.0.

Listing 3–5 LotteryOdds.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates a <code>for</code> loop.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */
8. public class LotteryOdds
9. {
10.    public static void main(String[] args)
11.    {
12.       Scanner in = new Scanner(System.in);
13.
14.       System.out.print("How many numbers do you need to draw? ");
15.       int k = in.nextInt();
16.
17.       System.out.print("What is the highest number you can draw? ");
18.       int n = in.nextInt();
19.
20.       /*
21.        * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
22.        */
23.
24.       int lotteryOdds = 1;
25.       for (int i = 1; i <= k; i++)
26.          lotteryOdds = lotteryOdds * (n - i + 1) / i;
27.
28.       System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
29.    }
30. }
```

Multiple Selections—The `switch` *Statement*

The `if/else` construct can be cumbersome when you have to deal with multiple selections with many alternatives. Java has a `switch` statement that is exactly like the `switch` statement in C and C++, warts and all.

For example, if you set up a menuing system with four alternatives like that in Figure 3–13, you could use code that looks like this:

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
   case 1:
      . . .
      break;
   case 2:
      . . .
      break;
   case 3:
      . . .
      break;
   case 4:
      . . .
      break;
   default:
      // bad input
      . . .
      break;
}
```

Execution starts at the `case` label that matches the value on which the selection is performed and continues until the next `break` or the end of the switch. If none of the case labels match, then the `default` clause is executed, if it is present.

CAUTION: It is possible for multiple alternatives to be triggered. If you forget to add a `break` at the end of an alternative, then execution falls through to the next alternative! This behavior is plainly dangerous and a common cause for errors. For that reason, we never use the `switch` statement in our programs.

The `case` labels must be integers or enumerated constants. You cannot test strings. For example, the following is an error:

```
String input = . . .;
switch (input) // ERROR
{
   case "A": // ERROR
      . . .
      break;
   . . .
}
```

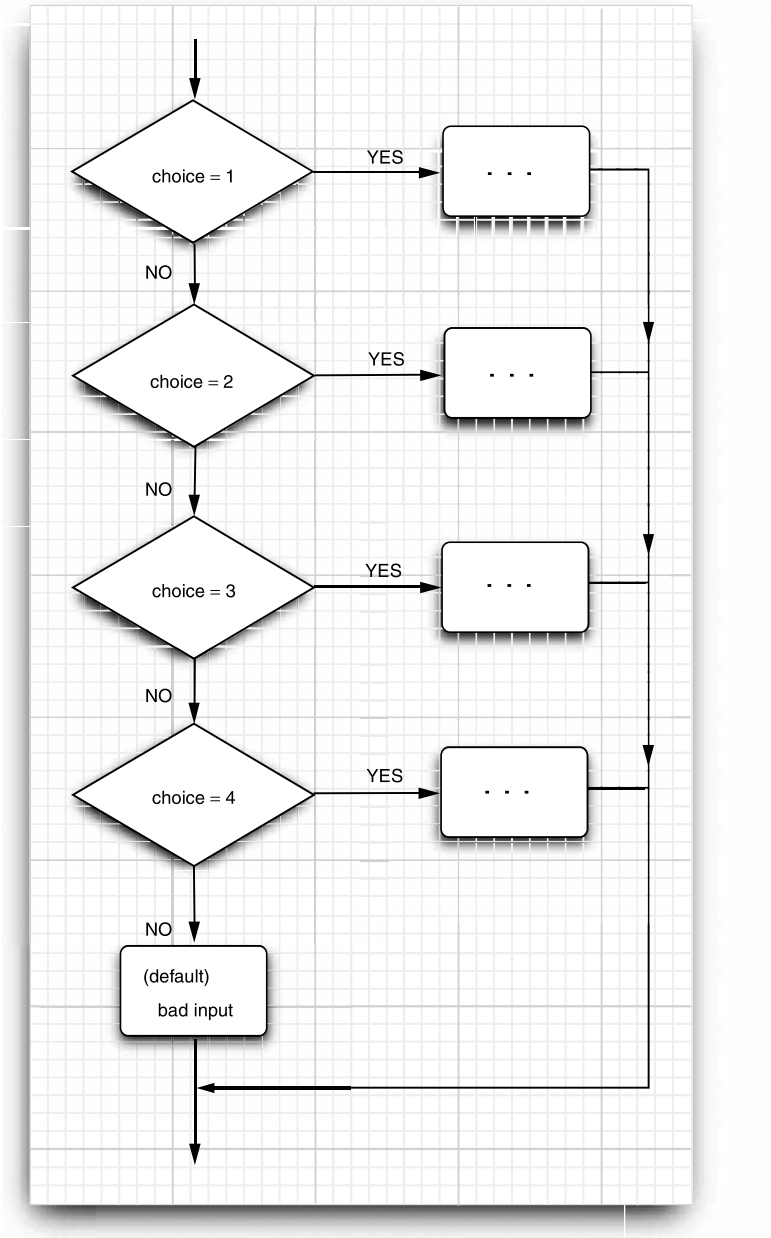

Figure 3–13 Flowchart for the `switch` statement

When you use the switch statement with enumerated constants, you need not supply the name of the enumeration in each label—it is deduced from the switch value. For example:

```
Size sz = . . .;
switch (sz)
{
   case SMALL: // no need to use Size.SMALL
      . . .
      break;
   . . .
}
```

Statements That Break Control Flow

Although the designers of Java kept the goto as a reserved word, they decided not to include it in the language. In general, goto statements are considered poor style. Some programmers feel the anti-goto forces have gone too far (see, for example, the famous article of Donald Knuth called "Structured Programming with goto statements"). They argue that unrestricted use of goto is error prone but that an occasional jump *out of a loop* is beneficial. The Java designers agreed and even added a new statement, the labeled break, to support this programming style.

Let us first look at the unlabeled break statement. The same break statement that you use to exit a switch can also be used to break out of a loop. For example:

```
while (years <= 100)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   if (balance >= goal) break;
   years++;
}
```

Now the loop is exited if either years > 100 occurs at the top of the loop or balance >= goal occurs in the middle of the loop. Of course, you could have computed the same value for years without a break, like this:

```
while (years <= 100 && balance < goal)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   if (balance < goal)
      years++;
}
```

But note that the test balance < goal is repeated twice in this version. To avoid this repeated test, some programmers prefer the break statement.

Unlike C++, Java also offers a *labeled break* statement that lets you break out of multiple nested loops. Occasionally something weird happens inside a deeply nested loop. In that case, you may want to break completely out of all the nested loops. It is inconvenient to program that simply by adding extra conditions to the various loop tests.

Here's an example that shows the break statement at work. Notice that the label must precede the outermost loop out of which you want to break. It also must be followed by a colon.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .) // this loop statement is tagged with the label
{
   . . .
   for (. . .) // this inner loop is not labeled
   {
      System.out.print("Enter a number >= 0: ");
      n = in.nextInt();
      if (n < 0) // should never happen-can't go on
         break read_data;
         // break out of read_data loop
      . . .
   }
}
// this statement is executed immediately after the labeled break
if (n < 0) // check for bad situation
{
   // deal with bad situation
}
else
{
   // carry out normal processing
}
```

If there was a bad input, the labeled break moves past the end of the labeled block. As with any use of the `break` statement, you then need to test whether the loop exited normally or as a result of a break.

NOTE: Curiously, you can apply a label to any statement, even an `if` statement or a block statement, like this:

```
label:
{
   . . .
   if (condition) break label; // exits block
   . . .
}
// jumps here when the break statement executes
```

Thus, if you are lusting after a `goto` and if you can place a block that ends just before the place to which you want to jump, you can use a `break` statement! Naturally, we don't recommend this approach. Note, however, that you can only jump *out of* a block, never *into* a block.

Finally, there is a `continue` statement that, like the `break` statement, breaks the regular flow of control. The `continue` statement transfers control to the header of the innermost enclosing loop. Here is an example:

```
Scanner in = new Scanner(System.in);
while (sum < goal)
{
   System.out.print("Enter a number: ");
   n = in.nextInt();
   if (n < 0) continue;
   sum += n; // not executed if n < 0
}
```

If `n < 0`, then the `continue` statement jumps immediately to the loop header, skipping the remainder of the current iteration.

If the `continue` statement is used in a `for` loop, it jumps to the "update" part of the `for` loop. For example, consider this loop:

```
for (count = 1; count <= 100; count++)
{
   System.out.print("Enter a number, -1 to quit: ");
   n = in.nextInt();
   if (n < 0) continue;
   sum += n; // not executed if n < 0
}
```

If `n < 0`, then the `continue` statement jumps to the `count++` statement.

There is also a labeled form of the `continue` statement that jumps to the header of the loop with the matching label.

TIP: Many programmers find the `break` and `continue` statements confusing. These statements are entirely optional—you can always express the same logic without them. In this book, we never use `break` or `continue`.

Big Numbers

If the precision of the basic integer and floating-point types is not sufficient, you can turn to a couple of handy classes in the `java.math` package: `BigInteger` and `BigDecimal`. These are classes for manipulating numbers with an arbitrarily long sequence of digits. The `BigInteger` class implements arbitrary precision integer arithmetic, and `BigDecimal` does the same for floating-point numbers.

Use the static `valueOf` method to turn an ordinary number into a big number:

```
BigInteger a = BigInteger.valueOf(100);
```

Unfortunately, you cannot use the familiar mathematical operators such as + and * to combine big numbers. Instead, you must use methods such as `add` and `multiply` in the big number classes.

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```

C++ NOTE: Unlike C++, Java has no programmable operator overloading. There was no way for the programmer of the `BigInteger` class to redefine the + and * operators to give the `add` and `multiply` operations of the `BigInteger` classes. The language designers did overload the + operator to denote concatenation of strings. They chose not to overload other operators, and they did not give Java programmers the opportunity to overload operators in their own classes.

Listing 3–6 shows a modification of the lottery odds program of Listing 3–5, updated to work with big numbers. For example, if you are invited to participate in a lottery in which you need to pick 60 numbers out of a possible 490 numbers, then this program will tell you that your odds are 1 in 7163958434619955574151162225400929334117176 12789263493493351 013459481104668848. Good luck!

The program in Listing 3–5 computed the statement

```
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

When big numbers are used, the equivalent statement becomes

```
lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(BigInteger.valueOf(i));
```

Listing 3–6 BigIntegerTest.java

```
1. import java.math.*;
2. import java.util.*;
3.
4. /**
5.  * This program uses big numbers to compute the odds of winning the grand prize in a lottery.
6.  * @version 1.20 2004-02-10
7.  * @author Cay Horstmann
8.  */
9. public class BigIntegerTest
10. {
11.    public static void main(String[] args)
12.    {
13.       Scanner in = new Scanner(System.in);
14.
15.       System.out.print("How many numbers do you need to draw? ");
16.       int k = in.nextInt();
17.
18.       System.out.print("What is the highest number you can draw? ");
19.       int n = in.nextInt();
20.
21.       /*
22.        * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23.        */
24.
25.       BigInteger lotteryOdds = BigInteger.valueOf(1);
26.
27.       for (int i = 1; i <= k; i++)
28.          lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(
29.                BigInteger.valueOf(i));
30.
31.       System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
32.    }
33. }
```

API `java.math.BigInteger` 1.1

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`

  returns the sum, difference, product, quotient, and remainder of this big integer and `other`.
- `int compareTo(BigInteger other)`

  returns 0 if this big integer equals `other`, a negative result if this big integer is less than `other`, and a positive result otherwise.
- `static BigInteger valueOf(long x)`

  returns a big integer whose value equals `x`.

API `java.math.BigDecimal` 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other, RoundingMode mode)` 5.0

  returns the sum, difference, product, or quotient of this big decimal and `other`. To compute the quotient, you must supply a *rounding mode*. The mode `RoundingMode.HALF_UP` is the rounding mode that you learned in school (i.e., round down digits 0 . . . 4, round up digits 5 . . . 9). It is appropriate for routine calculations. See the API documentation for other rounding modes.
- `int compareTo(BigDecimal other)`

  returns 0 if this big decimal equals `other`, a negative result if this big decimal is less than `other`, and a positive result otherwise.
- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x, int scale)`

  returns a big decimal whose value equals `x` or $x / 10^{scale}$.

Arrays

An array is a data structure that stores a collection of values of the same type. You access each individual value through an integer *index*. For example, if `a` is an array of integers, then `a[i]` is the `i`th integer in the array.

You declare an array variable by specifying the array type—which is the element type followed by `[]`—and the array variable name. For example, here is the declaration of an array `a` of integers:

```
int[] a;
```

However, this statement only declares the variable `a`. It does not yet initialize `a` with an actual array. You use the `new` operator to create the array.

```
int[] a = new int[100];
```

This statement sets up an array that can hold 100 integers.

NOTE: You can define an array variable either as

```
int[] a;
```

or as

```
int a[];
```

Most Java programmers prefer the former style because it neatly separates the type `int[]` (integer array) from the variable name.

The array entries are *numbered from 0 to 99* (and not 1 to 100). Once the array is created, you can fill the entries in an array, for example, by using a loop:

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
   a[i] = i;  // fills the array with 0 to 99
```

CAUTION: If you construct an array with 100 elements and then try to access the element `a[100]` (or any other index outside the range 0 . . . 99), then your program will terminate with an "array index out of bounds" exception.

To find the number of elements of an array, use *array*`.length`. For example:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Once you create an array, you cannot change its size (although you can, of course, change an individual array element). If you frequently need to expand the size of an array while a program is running, you should use a different data structure called an *array list*. (See Chapter 5 for more on array lists.)

The "for each" Loop

Java SE 5.0 introduced a powerful looping construct that allows you to loop through each element in an array (as well as other collections of elements) without having to fuss with index values.

The *enhanced* `for` loop

```
for (variable : collection) statement
```

sets the given variable to each element of the collection and then executes the statement (which, of course, may be a block). The *collection* expression must be an array or an object of a class that implements the `Iterable` interface, such as `ArrayList`. We discuss array lists in Chapter 5 and the `Iterable` interface in Chapter 2 of Volume II.

For example,

```
for (int element : a)
   System.out.println(element);
```

prints each element of the array `a` on a separate line.

You should read this loop as "for each `element` in `a`". The designers of the Java language considered using keywords such as `foreach` and `in`. But this loop was a late addition to the Java language, and in the end nobody wanted to break old code that already contains methods or variables with the same names (such as `System.in`).

Of course, you could achieve the same effect with a traditional `for` loop:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

However, the "for each" loop is more concise and less error prone. (You don't have to worry about those pesky start and end index values.)

NOTE: The loop variable of the "for each" loop traverses the *elements* of the array, not the index values.

The "for each" loop is a pleasant improvement over the traditional loop if you need to process all elements in a collection. However, there are still plenty of opportunities to use the traditional `for` loop. For example, you may not want to traverse the entire collection, or you may need the index value inside the loop.

TIP: There is an even easier way to print all values of an array, using the `toString` method of the `Arrays` class. The call `Arrays.toString(a)` returns a string containing the array elements, enclosed in brackets and separated by commas, such as `"[2, 3, 5, 7, 11, 13]"`. To print the array, simply call

```
System.out.println(Arrays.toString(a));
```

Array Initializers and Anonymous Arrays

Java has a shorthand to create an array object and supply initial values at the same time. Here's an example of the syntax at work:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

Notice that you do not call `new` when you use this syntax.

You can even initialize an *anonymous array:*

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

This expression allocates a new array and fills it with the values inside the braces. It counts the number of initial values and sets the array size accordingly. You can use this syntax to reinitialize an array without creating a new variable. For example,

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

is shorthand for

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```

NOTE: It is legal to have arrays of length 0. Such an array can be useful if you write a method that computes an array result and the result happens to be empty. You construct an array of length 0 as

```
new elementType[0]
```

Note that an array of length 0 is not the same as `null`. (See Chapter 4 for more information about `null`.)

Array Copying

You can copy one array variable into another, but then *both variables refer to the same array:*

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```

Figure 3–14 shows the result. If you actually want to copy all values of one array into a new array, you use the `copyTo` method in the `Arrays` class:

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

The second parameter is the length of the new array. A common use of this method is to increase the size of an array:

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

The additional elements are filled with 0 if the array contains numbers, `false` if the array contains `boolean` values. Conversely, if the length is less than the length of the original array, only the initial values are copied.

NOTE: Prior to Java SE 6, the `arraycopy` method in the `System` class was used to copy elements from one array to another. The syntax for this call is

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

The `to` array must have sufficient space to hold the copied elements.

Figure 3–14 Copying an array variable

For example, the following statements, whose result is illustrated in Figure 3–15, set up two arrays and then copy the last four entries of the first array to the second array. The copy starts at position 2 in the source array and copies four entries, starting at position 3 of the target.

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smallPrimes, 2, luckyNumbers, 3, 4);
for (int i = 0; i < luckyNumbers.length; i++)
   System.out.println(i + ": " + luckyNumbers[i]);
```

The output is

```
0: 1001
1: 1002
2: 1003
3: 5
4: 7
5: 11
6: 13
```

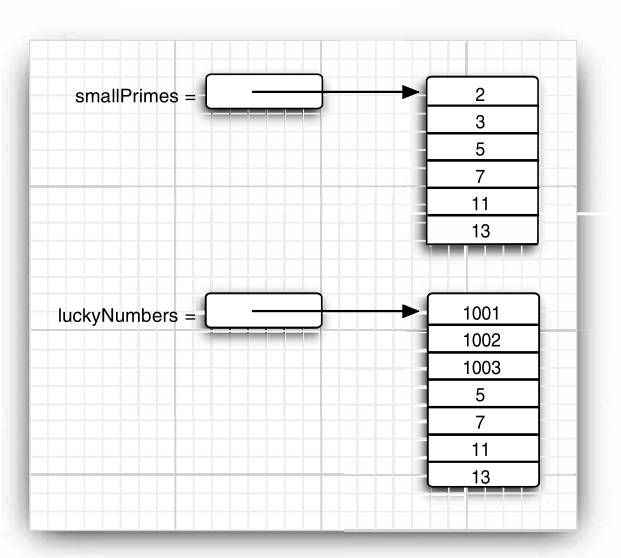

Figure 3–15 Copying values between arrays

C++ NOTE: A Java array is quite different from a C++ array on the stack. It is, however, essentially the same as a pointer to an array allocated on the *heap*. That is,

```
int[] a = new int[100]; // Java
```

is not the same as

```
int a[100]; // C++
```

but rather

```
int* a = new int[100]; // C++
```

In Java, the `[]` operator is predefined to perform *bounds checking*. Furthermore, there is no pointer arithmetic—you can't increment `a` to point to the next element in the array.

Command-Line Parameters

You have already seen one example of Java arrays repeated quite a few times. Every Java program has a `main` method with a `String[] args` parameter. This parameter indicates that the `main` method receives an array of strings, namely, the arguments specified on the command line.

For example, consider this program:

```
public class Message
{
   public static void main(String[] args)
   {
      if (args[0].equals("-h"))
         System.out.print("Hello,");
      else if (args[0].equals("-g"))
         System.out.print("Goodbye,");
      // print the other command-line arguments
      for (int i = 1; i < args.length; i++)
         System.out.print(" " + args[i]);
      System.out.println("!");
   }
}
```

If the program is called as

```
java Message -g cruel world
```

then the `args` array has the following contents:

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

The program prints the message

```
Goodbye, cruel world!
```

C++ NOTE: In the `main` method of a Java program, the name of the program is not stored in the `args` array. For example, when you start up a program as

```
java Message -h world
```

from the command line, then `args[0]` will be `"-h"` and not `"Message"` or `"java"`.

Array Sorting

To sort an array of numbers, you can use one of the `sort` methods in the `Arrays` class:

```
int[] a = new int[10000];
. . .
Arrays.sort(a)
```

This method uses a tuned version of the QuickSort algorithm that is claimed to be very efficient on most data sets. The `Arrays` class provides several other convenience methods for arrays that are included in the API notes at the end of this section.

The program in Listing 3–7 puts arrays to work. This program draws a random combination of numbers for a lottery game. For example, if you play a "choose 6 numbers from 49" lottery, then the program might print this:

```
Bet the following combination. It'll make you rich!
   4
   7
   8
   19
   30
   44
```

To select such a random set of numbers, we first fill an array `numbers` with the values 1, 2, . . . , `n`:

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
   numbers[i] = i + 1;
```

A second array holds the numbers to be drawn:

```
int[] result = new int[k];
```

Now we draw `k` numbers. The `Math.random` method returns a random floating-point number that is between 0 (inclusive) and 1 (exclusive). By multiplying the result with `n`, we obtain a random number between 0 and `n - 1`.

```
int r = (int) (Math.random() * n);
```

We set the `i`th result to be the number at that index. Initially, that is just `r + 1`, but as you'll see presently, the contents of the `numbers` array are changed after each draw.

```
result[i] = numbers[r];
```

Now we must be sure never to draw that number again—all lottery numbers must be distinct. Therefore, we overwrite `numbers[r]` with the *last* number in the array and reduce `n` by 1.

```
numbers[r] = numbers[n - 1];
n--;
```

The point is that in each draw we pick an *index,* not the actual value. The index points into an array that contains the values that have not yet been drawn.

After drawing `k` lottery numbers, we sort the `result` array for a more pleasing output:

```
Arrays.sort(result);
for (int r : result)
   System.out.println(r);
```

Listing 3–7 LotteryDrawing.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates array manipulation.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */
```

Listing 3–7 LotteryDrawing.java (continued)

```
8. public class LotteryDrawing
9. {
10.    public static void main(String[] args)
11.    {
12.       Scanner in = new Scanner(System.in);
13.
14.       System.out.print("How many numbers do you need to draw? ");
15.       int k = in.nextInt();
16.
17.       System.out.print("What is the highest number you can draw? ");
18.       int n = in.nextInt();
19.
20.       // fill an array with numbers 1 2 3 . . . n
21.       int[] numbers = new int[n];
22.       for (int i = 0; i < numbers.length; i++)
23.          numbers[i] = i + 1;
24.
25.       // draw k numbers and put them into a second array
26.       int[] result = new int[k];
27.       for (int i = 0; i < result.length; i++)
28.       {
29.          // make a random index between 0 and n - 1
30.          int r = (int) (Math.random() * n);
31.
32.          // pick the element at the random location
33.          result[i] = numbers[r];
34.
35.          // move the last element into the random location
36.          numbers[r] = numbers[n - 1];
37.          n--;
38.       }
39.
40.       // print the sorted array
41.       Arrays.sort(result);
42.       System.out.println("Bet the following combination. It'll make you rich!");
43.       for (int r : result)
44.          System.out.println(r);
45.    }
46. }
```

API java.util.Arrays 1.2

- static String toString(*type*[] a) 5.0
  returns a string with the elements of a, enclosed in brackets and delimited by commas.

  Parameters: a an array of type int, long, short, char, byte, boolean, float, or double.

- `static type copyOf(type[] a, int length)` 6
- `static type copyOf(type[] a, int start, int end)` 6

returns an array of the same type as `a`, of length either `length` or `end - start`, filled with the values of `a`.

Parameters:		
	`a`	an array of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.
	`start`	the starting index (inclusive).
	`end`	the ending index (exclusive). May be larger than `a.length`, in which case the result is padded with 0 or `false` values.
	`length`	the length of the copy. If `length` is larger than `a.length`, the result is padded with 0 or `false` values. Otherwise, only the initial `length` values are copied.

- `static void sort(type[] a)`

sorts the array, using a tuned QuickSort algorithm.

Parameters:		
	`a`	an array of type `int`, `long`, `short`, `char`, `byte`, `float`, or `double`.

- `static int binarySearch(type[] a, type v)`
- `static int binarySearch(type[] a, int start, int end type v)` 6

uses the binary search algorithm to search for the value `v`. If it is found, its index is returned. Otherwise, a negative value `r` is returned; `-r - 1` is the spot at which `v` should be inserted to keep `a` sorted.

Parameters:		
	`a`	a *sorted* array of type `int`, `long`, `short`, `char`, `byte`, `float`, or `double`.
	`start`	the starting index (inclusive).
	`end`	the ending index (exclusive).
	`v`	a value of the same type as the elements of `a`.

- `static void fill(type[] a, type v)`

sets all elements of the array to `v`.

Parameters:		
	`a`	an array of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.
	`v`	a value of the same type as the elements of `a`.

- `static boolean equals(type[] a, type[] b)`

returns `true` if the arrays have the same length, and if the elements in corresponding indexes match.

Parameters:		
	`a, b`	arrays of type `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

API java.lang.System 1.1

- `static void arraycopy(Object from, int fromIndex, Object to, int toIndex, int count)`
  copies elements from the first array to the second array.

  Parameters:

`from`	an array of any type (Chapter 5 explains why this is a parameter of type `Object`).
`fromIndex`	the starting index from which to copy elements.
`to`	an array of the same type as `from`.
`toIndex`	the starting index to which to copy elements.
`count`	the number of elements to copy.

Multidimensional Arrays

Multidimensional arrays use more than one index to access array elements. They are used for tables and other more complex arrangements. You can safely skip this section until you have a need for this storage mechanism.

Suppose you want to make a table of numbers that shows how much an investment of $10,000 will grow under different interest rate scenarios in which interest is paid annually and reinvested. Table 3–8 illustrates this scenario.

Table 3–8 Growth of an Investment at Different Interest Rates

10%	11%	12%	13%	14%	15%
10,000.00	10,000.00	10,000.00	10,000.00	10,000.00	10,000.00
11,000.00	11,100.00	11,200.00	11,300.00	11,400.00	11,500.00
12,100.00	12,321.00	12,544.00	12,769.00	12,996.00	13,225.00
13,310.00	13,676.31	14,049.28	14,428.97	14,815.44	15,208.75
14,641.00	15,180.70	15,735.19	16,304.74	16,889.60	17,490.06
16,105.10	16,850.58	17,623.42	18,424.35	19,254.15	20,113.57
17,715.61	18,704.15	19,738.23	20,819.52	21,949.73	23,130.61
19,487.17	20,761.60	22,106.81	23,526.05	25,022.69	26,600.20
21,435.89	23,045.38	24,759.63	26,584.44	28,525.86	30,590.23
23,579.48	25,580.37	27,730.79	30,040.42	32,519.49	35,178.76

You can store this information in a two-dimensional array (or matrix), which we call `balances`.

Declaring a two-dimensional array in Java is simple enough. For example:

```
double[][] balances;
```

As always, you cannot use the array until you initialize it with a call to `new`. In this case, you can do the initialization as follows:

```
balances = new double[NYEARS][NRATES];
```

In other cases, if you know the array elements, you can use a shorthand notion for initializing multidimensional arrays without needing a call to `new`. For example:

```
int[][] magicSquare =
   {
      {16, 3, 2, 13},
      {5, 10, 11, 8},
      {9, 6, 7, 12},
      {4, 15, 14, 1}
   };
```

Once the array is initialized, you can access individual elements by supplying two brackets, for example, `balances[i][j]`.

The example program stores a one-dimensional array `interest` of interest rates and a two-dimensional array `balance` of account balances, one for each year and interest rate. We initialize the first row of the array with the initial balance:

```
for (int j = 0; j < balance[0].length; j++)
   balances[0][j] = 10000;
```

Then we compute the other rows, as follows:

```
for (int i = 1; i < balances.length; i++)
{
   for (int j = 0; j < balances[i].length; j++)
   {
      double oldBalance = balances[i - 1][j];
      double interest = . . .;
      balances[i][j] = oldBalance + interest;
   }
}
```

Listing 3–8 shows the full program.

NOTE: A "for each" loop does not automatically loop through all entries in a two-dimensional array. Instead, it loops through the rows, which are themselves one-dimensional arrays. To visit all elements of a two-dimensional array `a`, nest two loops, like this:

```
for (double[] row : a)
   for (double value : row)
      do something with value
```

TIP: To print out a quick and dirty list of the elements of a two-dimensional array, call

```
System.out.println(Arrays.deepToString(a));
```

The output is formatted like this:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

Listing 3–8 CompoundInterest.java

```
/**
 * This program shows how to store tabular data in a 2D array.
 * @version 1.40 2004-02-10
 * @author Cay Horstmann
 */
public class CompoundInterest
{
   public static void main(String[] args)
   {
      final double STARTRATE = 10;
      final int NRATES = 6;
      final int NYEARS = 10;

      // set interest rates to 10 . . . 15%
      double[] interestRate = new double[NRATES];
      for (int j = 0; j < interestRate.length; j++)
         interestRate[j] = (STARTRATE + j) / 100.0;

      double[][] balances = new double[NYEARS][NRATES];

      // set initial balances to 10000
      for (int j = 0; j < balances[0].length; j++)
         balances[0][j] = 10000;

      // compute interest for future years
      for (int i = 1; i < balances.length; i++)
      {
         for (int j = 0; j < balances[i].length; j++)
         {
            // get last year's balances from previous row
            double oldBalance = balances[i - 1][j];

            // compute interest
            double interest = oldBalance * interestRate[j];

            // compute this year's balances
            balances[i][j] = oldBalance + interest;
         }
      }

      // print one row of interest rates
      for (int j = 0; j < interestRate.length; j++)
         System.out.printf("%9.0f%%", 100 * interestRate[j]);

      System.out.println();

      // print balance table
      for (double[] row : balances)
      {
```

Listing 3–8 CompoundInterest.java (continued)

```
50.          // print table row
51.          for (double b : row)
52.             System.out.printf("%10.2f", b);
53.
54.          System.out.println();
55.       }
56.    }
57. }
```

Ragged Arrays

So far, what you have seen is not too different from other programming languages. But there is actually something subtle going on behind the scenes that you can sometimes turn to your advantage: Java has *no* multidimensional arrays at all, only one-dimensional arrays. Multidimensional arrays are faked as "arrays of arrays."

For example, the `balances` array in the preceding example is actually an array that contains ten elements, each of which is an array of six floating-point numbers (see Figure 3–16).

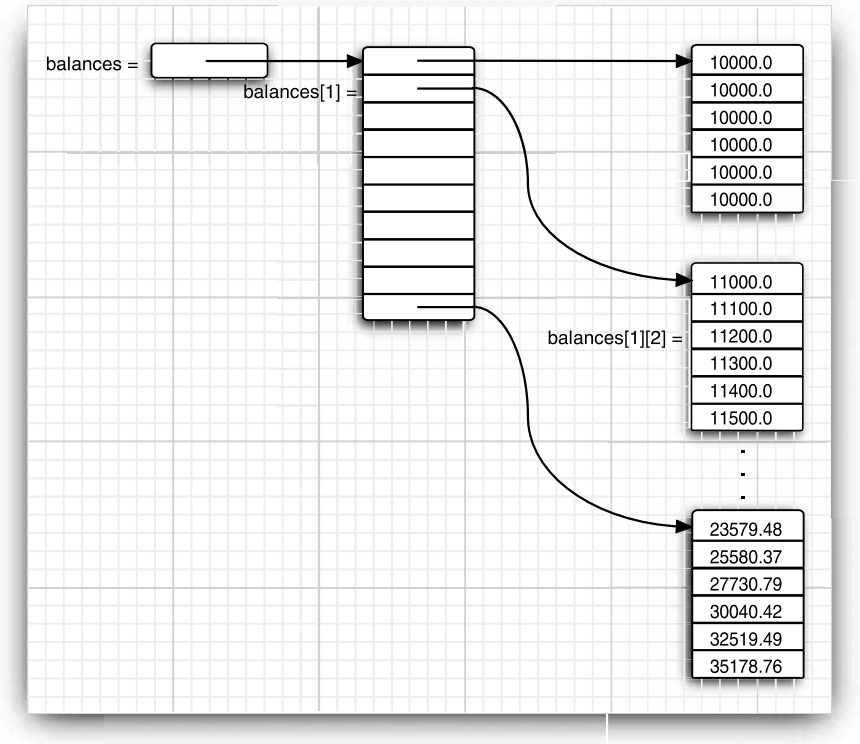

Figure 3–16 A two-dimensional array

The expression `balances[i]` refers to the `i`th subarray, that is, the `i`th row of the table. It is itself an array, and `balances[i][j]` refers to the `j`th entry of that array.

Because rows of arrays are individually accessible, you can actually swap them!

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

It is also easy to make "ragged" arrays, that is, arrays in which different rows have different lengths. Here is the standard example. Let us make an array in which the entry at row `i` and column `j` equals the number of possible outcomes of a "choose `j` numbers from `i` numbers" lottery.

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
```

Because `j` can never be larger than `i`, the matrix is triangular. The `i`th row has `i + 1` elements. (We allow choosing 0 elements; there is one way to make such a choice.) To build this ragged array, first allocate the array holding the rows.

```
int[][] odds = new int[NMAX + 1][];
```

Next, allocate the rows.

```
for (int n = 0; n <= NMAX; n++)
   odds[n] = new int[n + 1];
```

Now that the array is allocated, we can access the elements in the normal way, provided we do not overstep the bounds.

```
for (int n = 0; n < odds.length; n++)
   for (int k = 0; k < odds[n].length; k++)
   {
      // compute lotteryOdds
      . . .
      odds[n][k] = lotteryOdds;
   }
```

Listing 3–9 gives the complete program.

C++ NOTE: In C++, the Java declaration

```
double[][] balances = new double[10][6]; // Java
```

is not the same as

```
double balances[10][6]; // C++
```

or even

```
double (*balances)[6] = new double[10][6]; // C++
```

Instead, an array of 10 pointers is allocated:

```
double** balances = new double*[10]; // C++
```

Then, each element in the pointer array is filled with an array of 6 numbers:

```
for (i = 0; i < 10; i++)
   balances[i] = new double[6];
```

Mercifully, this loop is automatic when you ask for a `new double[10][6]`. When you want ragged arrays, you allocate the row arrays separately.

Listing 3–9 LotteryArray.java

```
/**
 * This program demonstrates a triangular array.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryArray
{
   public static void main(String[] args)
   {
      final int NMAX = 10;

      // allocate triangular array
      int[][] odds = new int[NMAX + 1][];
      for (int n = 0; n <= NMAX; n++)
         odds[n] = new int[n + 1];

      // fill triangular array
      for (int n = 0; n < odds.length; n++)
         for (int k = 0; k < odds[n].length; k++)
         {
            /*
             * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
             */
            int lotteryOdds = 1;
            for (int i = 1; i <= k; i++)
               lotteryOdds = lotteryOdds * (n - i + 1) / i;

            odds[n][k] = lotteryOdds;
         }

      // print triangular array
      for (int[] row : odds)
      {
         for (int odd : row)
            System.out.printf("%4d", odd);
         System.out.println();
      }
   }
}
```

You have now seen the fundamental programming structures of the Java language. The next chapter covers object-oriented programming in Java.