Chapter 1

In this chapter, you will learn about the basic data types and control structures of the Java language. I assume that you are an experienced programmer in some other language and that you are familiar with concepts such as variables, loops, function calls, and arrays, but perhaps with a different syntax. This chapter will get you up to speed on the Java way. I will also give you some tips on the most useful parts of the Java API for manipulating common data types.

The key points of this chapter are:

1. In Java, all methods are declared in a class. You invoke a nonstatic method on an object of the class to which the method belongs.
2. Static methods are not invoked on objects. Program execution starts with the static `main` method.
3. Java has eight primitive types: four signed integral types, two floating-point types, `char`, and `boolean`.
4. The Java operators and control structures are very similar to those of C or JavaScript.
5. The `Math` class provides common mathematical functions.
6. `String` objects are sequences of characters or, more precisely, Unicode code points in the UTF-16 encoding.

7. With the `System.out` object, you can display output in a terminal window. A `Scanner` tied to `System.in` lets you read terminal input.
8. Arrays and collections can be used to collect elements of the same type.

1.1 Our First Program

When learning any new programming language, it is traditional to start with a program that displays the message "Hello, World!". That is what we will do in the following sections.

1.1.1 Dissecting the "Hello, World" Program

Without further ado, here is the "Hello, World" program in Java.

```
package ch01.sec01;

// Our first Java program

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

Let's examine this program:

- Java is an object-oriented language. In your program, you manipulate (mostly) *objects* by having them do work. Each object that you manipulate belongs to a specific *class*, and we say that the object is an *instance* of that class. A class defines what an object's state can be and and what it can do. In Java, all code is defined inside classes. We will look at objects and classes in detail in Chapter 2. This program is made up of a single class `HelloWorld`.
- `main` is a *method*, that is, a function declared inside a class. The `main` method is the first method that is called when the program runs. It is declared as `static` to indicate that the method does not operate on any objects. (When `main` gets called, there are only a handful of predefined objects, and none of them are instances of the `HelloWorld` class.) The method is declared as `void` to indicate that it does not return any value. See Section 1.8.8, "Command-Line Arguments" (page 49) for the meaning of the parameter declaration `String[] args`.
- In Java, you can declare many features as `public` or `private`, and there are a couple of other visibility levels as well. Here, we declare the `HelloWorld`

class and the `main` method as `public`, which is the most common arrangement for classes and methods.

- A *package* is a set of related classes. It is a good idea to place each class in a package so you can group related classes together and avoid conflicts when multiple classes have the same name. In this book, we'll use chapter and section numbers as package names. The full name of our class is `ch01.sec01.HelloWorld`. Chapter 2 has more to say about packages and package naming conventions.
- The line starting with // is a comment. All characters between // and the end of the line are ignored by the compiler and are meant for human readers only.
- Finally, we come to the body of the `main` method. In our example, it consists of a single line with a command to print a message to `System.out`, an object representing the "standard output" of the Java program.

As you can see, Java is not a scripting language that can be used to quickly dash off a few commands. It is squarely intended as a language for larger programs that benefit from being organized into classes, packages, and modules. (Modules are introduced in Chapter 15.)

Java is also quite simple and uniform. Some languages have global variables and functions as well as variables and methods inside classes. In Java, everything is declared inside a class. This uniformity can lead to somewhat verbose code, but it makes it easy to understand the meaning of a program.

NOTE: You have just seen a // comment that extends to the end of the line. You can also have multiline comments between `/*` and `*/` delimiters, such as

```
/*
    This is the first sample program in Core Java for the Impatient.
    The program displays the traditional greeting "Hello, World!".
*/
```

There is a third comment style, called *documentation comment*, with `/**` and `*/` as delimiters, that you will see in the next chapter.

1.1.2 Compiling and Running a Java Program

To compile and run this program, you need to install the Java Development Kit (JDK) and, optionally, an integrated development environment (IDE). You should also download the sample code, which you will find at the companion website for this book, `http://horstmann.com/javaimpatient`. Since instructions for

installing software don't make for interesting reading, I put them on the companion website as well.

Once you have installed the JDK, open a terminal window, change to the directory containing the `ch01` directory, and run the commands

```
javac ch01/sec01/HelloWorld.java
java ch01.sec01.HelloWorld
```

The familiar greeting will appear in the terminal window (see Figure 1-1).

Note that two steps were involved to execute the program. The `javac` command *compiles* the Java source code into an intermediate machine-independent representation, called *byte codes*, and saves them in *class files*. The `java` command launches a *virtual machine* that loads the class files and executes the byte codes.

Once compiled, byte codes can run on any Java virtual machine, whether on your desktop computer or on a device in a galaxy far, far away. The promise of "write once, run anywhere" was an important design criterion for Java.

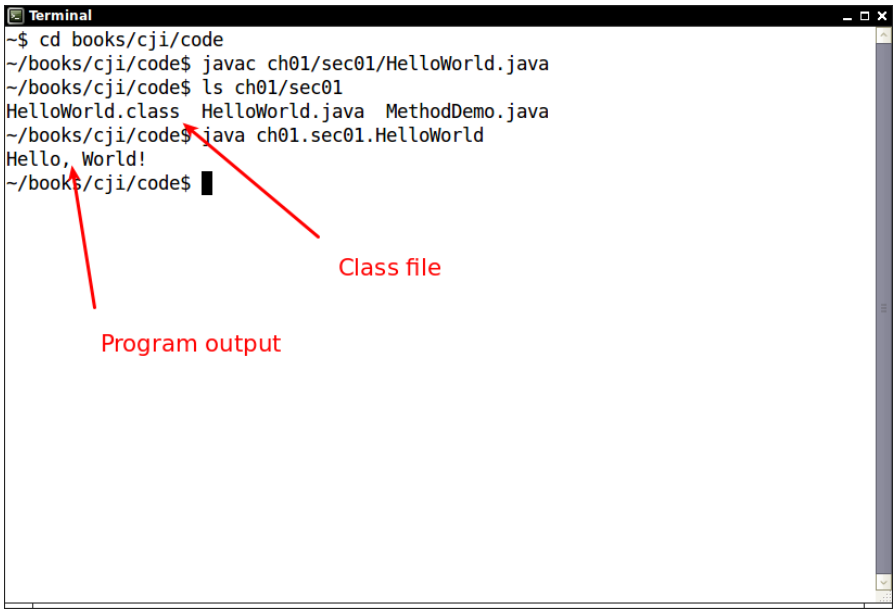

Figure 1-1 Running a Java program in a terminal window

NOTE: The `javac` compiler is invoked with the name of a *file*, with slashes separating the path segments, and an extension `.java`. The `java` virtual machine launcher is invoked with the name of a *class*, with dots separating the package segments, and no extension.

To run the program in an IDE, you need to first make a project, as described in the installation instructions. Then, select the `HelloWorld` class and tell the IDE to run it. Figure 1-2 shows how this looks in Eclipse. Eclipse is a popular IDE, but there are many other excellent choices. As you get more comfortable with Java programming, you should try out a few and pick one that you like.

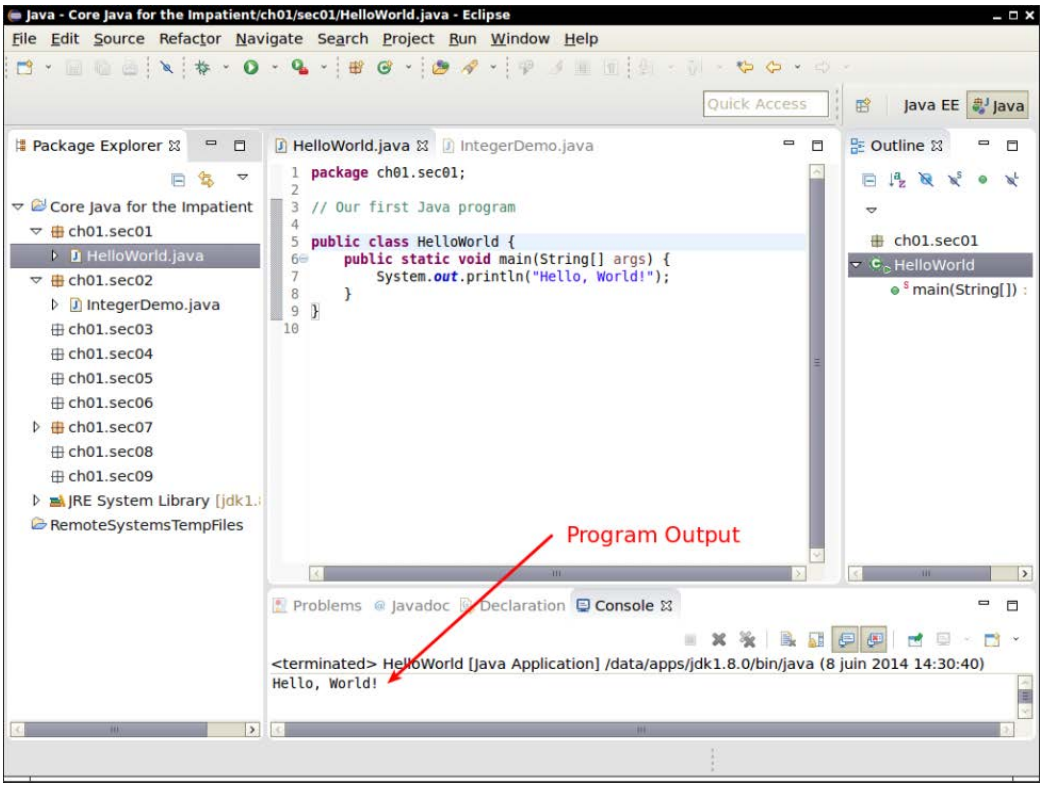

Figure 1-2 Running a Java program inside the Eclipse IDE

Congratulations! You have just followed the time-honored ritual of running the "Hello, World!" program in Java. Now we are ready to examine the basics of the Java language.

1.1.3 Method Calls

Let us have a closer look at the single statement of the `main` method:

```
System.out.println("Hello, World!");
```

`System.out` is an object. It is an *instance* of a class called `PrintStream`. The `PrintStream` class has methods `println`, `print`, and so on. These methods are called *instance methods* because they operate on objects, or instances, of the class.

To invoke an instance method on an object, you use the *dot notation*

object.*methodName*(*arguments*)

In this case, there is just one argument, the string `"Hello, World!"`.

Let's try it with another example. Strings such as `"Hello, World!"` are instances of the `String` class. The `String` class has a method `length` that returns the length of a `String` object. To call the method, you again use the dot notation:

```
"Hello, World!".length()
```

The `length` method is invoked on the object `"Hello, World!"`, and it has no arguments. Unlike the `println` method, the `length` method returns a result. One way of using that result is to print it:

```
System.out.println("Hello, World!".length());
```

Give it a try. Make a Java program with this statement and run it to see how long the string is.

In Java, you need to *construct* most objects (unlike the `System.out` and `"Hello, World!"` objects, which are already there, ready for you to use). Here is a simple example.

An object of the `Random` class can generate random numbers. You construct a `Random` object with the `new` operator:

```
new Random()
```

After the class name is the list of construction arguments, which is empty in this example.

You can call a method on the constructed object. The call

```
new Random().nextInt()
```

yields the next integer that the newly constructed random number generator has to offer.

If you want to invoke more than one method on an object, store it in a variable (see Section 1.3, "Variables," page 14). Here we print two random numbers:

```
Random generator = new Random();
System.out.println(generator.nextInt());
System.out.println(generator.nextInt());
```

NOTE: The `Random` class is declared in the `java.util` package. To use it in your program, add an `import` statement, like this:

```
package ch01.sec01;

import java.util.Random;

public class MethodDemo {
    ...
}
```

We will look at packages and the `import` statement in more detail in Chapter 2.

1.1.4 JShell

In Section 1.1.2, "Compiling and Running a Java Program" (page 3), you saw how to compile and run a Java program. Java 9 introduces another way of working with Java. The JShell program provides a "read-evaluate-print loop" (REPL) where you type a Java expression, JShell evaluates your input, prints the result, and waits for your next input. To start JShell, simply type `jshell` in a terminal window (Figure 1-3).

JShell starts with a greeting, followed by a prompt:

```
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro

jshell>
```

Now type any Java expression, such as

```
"Hello, World!".length()
```

JShell responds with the result and another prompt.

```
$1 ==> 13

jshell>
```

Note that you do *not* type `System.out.println`. JShell automatically prints the value of every expression that you enter.

The `$1` in the output indicates that the result is available in further calculations. For example, if you type

```
3 * $1 + 3
```

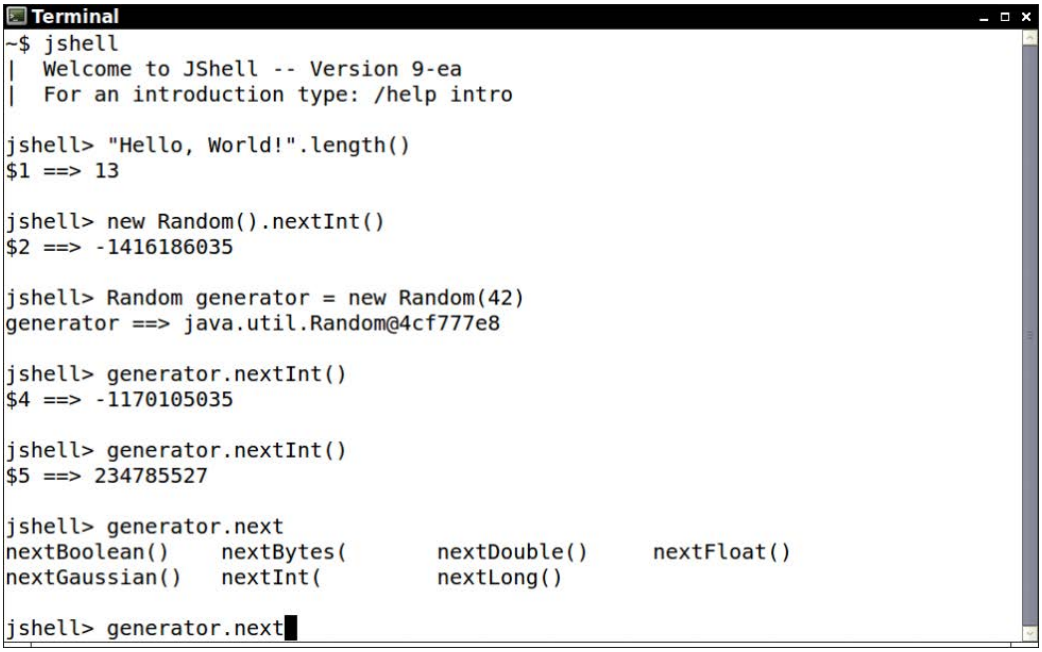

Figure 1-3 Running JShell

the response is

```
$2 ==> 42
```

If you need a variable many times, you can give it a more memorable name. You have to follow the Java syntax and specify both the type and the name (see Section 1.3, "Variables," page 14). For example,

```
jshell> int answer = 42
answer ==> 42
```

You can have JShell fill in the type for you. Type an expression and instead of hitting the Enter key, hit Shift+Tab and then the V key. For example, when you type

```
new Random()
```

followed by Shift+Tab and the V key, you get

```
jshell> Random  = new Random()
```

with the cursor positioned just before the = symbol. Now type a variable name and hit Enter:

```
jshell> Random generator = new Random()
generator ==> java.util.Random@3fee9989
```

Another useful feature is tab completion. Type

```
generator.
```

followed by the Tab key. You get a list of all methods that you can invoke on the `generator` variable:

```
jshell> generator.
doubles(         equals(          getClass()       hashCode()
ints(            longs(           nextBoolean()    nextBytes(
nextDouble()     nextFloat()      nextGaussian()   nextInt(
nextLong()       notify()         notifyAll()      setSeed(
toString()       wait(
```

Now type `ne` and hit the Tab key again. The method name is completed to `next`, and you get a shorter list:

```
jshell> generator.next
nextBoolean()    nextBytes(       nextDouble()     nextFloat()
nextGaussian()   nextInt(         nextLong()
```

Type a D and Tab again, and now the only completion, `nextDouble()`, is filled in. Hit Enter to accept it:

```
jshell> generator.nextDouble()
$8 ==> 0.9560346568377398
```

NOTE: Note that in the autocompletion list, methods that require an argument are only followed by a left parenthesis, such as `nextInt(`, but methods without arguments have both parentheses, such as `nextBoolean()`.

To repeat a command, hit the ↑ key until you see the line that you want to reissue or edit. You can move the cursor in the line with the ← and → keys, and add or delete characters. Hit Enter when you are done. For example, hit ↑ and replace `Double` with `Int`, then hit Enter:

```
jshell> generator.nextInt()
$9 ==> -352355569
```

By default, JShell imports the following packages:

```
java.io
java.math
java.net
java.nio.file
java.util
java.util.concurrent
java.util.function
java.util.prefs
java.util.regex
java.util.stream
```

That's why you can use the `Random` class in JShell without any import statements. If you need to import another class, you can type the import statement at the JShell prompt. Or, more conveniently, you can have JShell search for it, by typing Shift+Tab and the I key. For example, type `Duration` followed by Shift+Tab and the I key. You get a list of potential actions:

```
jshell> Duration
0: Do nothing
1: import: java.time.Duration
2: import: javafx.util.Duration
3: import: javax.xml.datatype.Duration
Choice:
```

Type `1`, and you receive a confirmation:

```
Imported: java.time.Duration
```

followed by

```
jshell> Duration
```

so that you can pick up where you left off, but with the import in place.

These commands are enough to get you started with JShell. To get help, type `/help` and Enter. To exit, type `/exit` and Enter, or simply Ctrl+D.

JShell makes it easy and fun to learn about the Java language and library, without having to launch a heavy-duty development environment, and without fussing with `public static void main`.

1.2 Primitive Types

Even though Java is an object-oriented programming language, not all Java values are objects. Instead, some values belong to *primitive types*. Four of these types are signed integer types, two are floating-point number types, one is the character type `char` that is used in the encoding for strings, and one is the `boolean` type for truth values. We will look at these types in the following sections.

1.2.1 Signed Integer Types

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

Table 1–1 Java Signed Integer Types

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

NOTE: The constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE` are the smallest and largest `int` values. The `Long`, `Short`, and `Byte` classes also have `MIN_VALUE` and `MAX_VALUE` constants.

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.

NOTE: If the `long` type is not sufficient, use the `BigInteger` class. See Section 1.4.6, "Big Numbers" (page 23) for details.

In Java, the ranges of the integer types do not depend on the machine on which you will be running your program. After all, Java is designed as a "write once, run anywhere" language. In contrast, the integer types in C and C++ programs depend on the processor for which a program is compiled.

You write `long` integer literals with a suffix `L` (for example, `4000000000L`). There is no syntax for literals of type `byte` or `short`. Use the cast notation (see Section 1.4.4, "Number Type Conversions," page 20), for example, `(byte) 127`.

Hexadecimal literals have a prefix `0x` (for example, `0xCAFEBABE`). Binary values have a prefix `0b`. For example, `0b1001` is `9`.

CAUTION: Octal numbers have a prefix `0`. For example, `011` is `9`. This can be confusing, so it seems best to stay away from octal literals and leading zeroes.

You can add underscores to number literals, such as `1_000_000` (or `0b1111_0100_0010_0100_0000`) to denote one million. The underscores are for human eyes only, the Java compiler simply removes them.

NOTE: If you work with integer values that can never be negative and you really need an additional bit, you can, with some care, interpret signed integer values as unsigned. For example, a `byte` value `b` represents the range from –128 to 127. If you want a range from 0 to 255, you can still store it in a `byte`. Due to the nature of binary arithmetic, addition, subtraction, and multiplication will all work, provided they don't overflow. For other operations, call `Byte.toUnsignedInt(b)` to get an `int` value between 0 and 255, then process the integer value, and cast the result back to `byte`. The `Integer` and `Long` classes have methods for unsigned division and remainder.

1.2.2 Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in Table 1-2.

Table 1-2 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)

Many years ago, when memory was a scarce resource, four-byte floating-point numbers were in common use. But seven decimal digits don't go very far, and nowadays, "double precision" numbers are the default. It only makes sense to use `float` when you need to store a large number of them.

Numbers of type `float` have a suffix `F` (for example, `3.14F`). Floating-point literals without an `F` suffix (such as `3.14`) have type `double`. You can optionally supply the `D` suffix (for example, `3.14D`).

NOTE: You can specify floating-point literals in hexadecimal. For example, 0.0009765625 = 2^{-10} can be written as `0x1.0p-10`. In hexadecimal notation, you use a `p`, not an `e`, to denote the exponent. (An `e` is a hexadecimal digit.) Note that, even though the digits are written in hexadecimal, the exponent (that is, the power of 2) is written in decimal.

There are special floating-point values `Double.POSITIVE_INFINITY` for ∞, `Double.NEGATIVE_INFINITY` for −∞, and `Double.NaN` for "not a number." For example, the result of computing `1.0 / 0.0` is positive infinity. Computing `0.0 / 0.0` or the square root of a negative number yields NaN.

CAUTION: All "not a number" values are considered to be distinct from each other. Therefore, you cannot use the test `if (x == Double.NaN)` to check whether `x` is a NaN. Instead, call `if (Double.isNaN(x))`. There are also methods `Double.isInfinite` to test for ±∞, and `Double.isFinite` to check that a floating-point number is neither infinite nor a NaN.

Floating-point numbers are not suitable for financial calculations 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 with arbitrary precision and without roundoff errors, use the `BigDecimal` class, introduced in Section 1.4.6, "Big Numbers" (page 23).

1.2.3 The `char` Type

The `char` type describes "code units" in the UTF-16 character encoding used by Java. The details are rather technical—see Section 1.5, "Strings" (page 24). You probably won't use the `char` type very much.

Occasionally, you may encounter character literals, enclosed in single quotes. For example, `'J'` is a character literal with value 74 (or hexadecimal 4A), the code unit for denoting the Unicode character "U+004A Latin Capital Letter J." A code unit can be expressed in hexadecimal, with the `\u` prefix. For example, `'\u004A'` is the same as `'J'`. A more exotic example is `'\u263A'`, the code unit for ☺, "U+263A White Smiling Face."

The special codes '\n', '\r', '\t', '\b' denote a line feed, carriage return, tab, and backspace.

Use a backslash to escape a single quote '\'' and a backslash '\\'.

1.2.4 The boolean Type

The boolean type has two values, false and true.

In Java, the boolean type is not a number type. There is no relationship between boolean values and the integers 0 and 1.

1.3 Variables

In the following sections, you will learn how to declare and initialize variables and constants.

1.3.1 Variable Declarations

Java is a strongly typed language. Each variable can only hold values of a specific type. When you declare a variable, you need to specify the type, the name, and an optional initial value. For example,

```
int total = 0;
```

You can declare multiple variables of the same type in a single statement:

```
int total = 0, count; // count is an uninitialized integer
```

Most Java programmers prefer to use separate declarations for each variable.

Consider this variable declation:

```
Random generator = new Random();
```

Here, the name of the object's class occurs twice. The first Random is the type of the variable generator. The second Random is a part of the new expression for constructing an object of that class.

1.3.2 Names

The name of a variable (as well as a method or class) must begin with a letter. It can consist of any letters, digits, and the symbols _ and $. However, the $ symbol is intended for automatically generated names, and you should not use it in your names. Finally, the _ by itself is not a valid variable name.

Here, letters and digits can be from *any* alphabet, not just the Latin alphabet. For example, `π` and `élévation` are valid variable names. Letter case is significant: `count` and `Count` are different names.

You cannot use spaces or symbols in a name. Finally, you cannot use a keyword such as `double` as a name.

By convention, names of variables and methods start with a lowercase letter, and names of classes start with an uppercase letter. Java programmers like "camel case," where uppercase letters are used when names consist of multiple words, like `countOfInvalidInputs`.

1.3.3 Initialization

When you declare a variable in a method, you must initialize it before you can use it. For example, the following code results in a compile-time error:

```
int count;
count++; // Error—uses an uninitialized variable
```

The compiler must be able to verify that a variable has been initialized before it has been used. For example, the following code is also an error:

```
int count;
if (total == 0) {
    count = 0;
} else {
    count++; // Error—count might not be initialized
}
```

You are allowed to declare a variable anywhere within a method. It is considered good style to declare a variable as late as possible, just before you need it for the first time. For example,

```
Scanner in = new Scanner(System.in); // See Section 1.6.1 for reading input
System.out.println("How old are you?");
int age = in.nextInt();
```

The variable is declared at the point at which its initial value is available.

1.3.4 Constants

The `final` keyword denotes a value that cannot be changed once it has been assigned. In other languages, one would call such a value a *constant*. For example,

```
final int DAYS_PER_WEEK = 7;
```

By convention, uppercase letters are used for names of constants.

You can also declare a constant outside a method, using the `static` keyword:

```
public class Calendar {
    public static final int DAYS_PER_WEEK = 7;
    ...
}
```

Then the constant can be used in multiple methods. Inside `Calendar`, you refer to the constant as `DAYS_PER_WEEK`. To use the constant in another class, prepend the class name: `Calendar.DAYS_PER_WEEK`.

NOTE: The `System` class declares a constant

```
public static final PrintStream out
```

that you can use anywhere as `System.out`. This is one of the few examples of a constant that is not written in uppercase.

It is legal to defer the initialization of a `final` variable, provided it is initialized exactly once before it is used for the first time. For example, the following is legal:

```
final int DAYS_IN_FEBRUARY;
if (leapYear) {
    DAYS_IN_FEBRUARY = 29;
} else {
    DAYS_IN_FEBRUARY = 28;
}
```

That is the reason for calling them "final" variables. Once a value has been assigned, it is final and can never be changed.

NOTE: Sometimes, you need a set of related constants, such as

```
public static final int MONDAY = 0;
public static final int TUESDAY = 1;
...
```

In this case, you can define an *enumerated type* like this:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY };
```

Then, `Weekday` is a type with values `Weekday.MONDAY` and so on. Here is how you declare and initialize a `Weekday` variable:

```
Weekday startDay = Weekday.MONDAY;
```

We will discuss enumerated types in Chapter 4.

1.4 Arithmetic Operations

Java uses the familiar operators of any C-based language (see Table 1-3). We will review them in the following sections.

Table 1-3 Java Operators

Operators	Associativity
`[] . ()` (method call)	Left
`! ~ ++ -- +` (unary) `-` (unary) `()` (cast) `new`	Right
`* / %` (modulus)	Left
`+ -`	Left
`<< >> >>>` (arithmetic shift)	Left
`< > <= >= instanceof`	Left
`== !=`	Left
`&` (bitwise and)	Left
`^` (bitwise exclusive or)	Left
`|` (bitwise or)	Left
`&&` (logical and)	Left
`||` (logical or)	Left
`? :` (conditional)	Left
`= += -= *= /= %= <<= >>= >>>= &= ^= |=`	Right

NOTE: In this table, operators are listed by decreasing *precedence*. For example, since `+` has a higher precedence than `<<`, the value of `3 + 4 << 5` is `(3 + 4) << 5`. An operator is *left-associative* when it is grouped left to right. For example, `3 - 4 - 5` means `(3 - 4) - 5`. But `-=` is right-associative, and `i -= j -= k` means `i -= (j -= k)`.

1.4.1 Assignment

The last row in Table 1-3 shows the assignment operators. The statement

```
x = expression;
```

sets `x` to the value of the right-hand side, replacing the previous value. When `=` is preceded by an operator, the operator combines the left- and right-hand sides and the result is assigned. For example,

```
amount -= 10;
```

is the same as

```
amount = amount - 10;
```

1.4.2 Basic Arithmetic

Addition, subtraction, multiplication, and division are denoted by `+ - * /`. For example, `2 * n + 1` means to multiply `2` and `n` and add `1`.

You need to be careful with the `/` operator. If both operands are integer types, it denotes integer division, discarding the remainder. For example, `17 / 5` is `3`, whereas `17.0 / 5` is `3.4`.

An integer division by zero gives rise to an exception which, if not caught, will terminate your program. (See Chapter 5 for more information on exception handling.) A floating-point division by zero yields an infinite value or NaN (see Section 1.2.2, "Floating-Point Types," page 12), without causing an exception.

The `%` operator yields the remainder. For example, `17 % 5` is `2`, the amount that remains from 17 after subtracting 15 (the largest integer multiple of 5 that "fits" into 17). If the remainder of `a % b` is zero, then `a` is an integer multiple of `b`.

A common use is to test whether an integer is even. The expression `n % 2` is `0` if `n` is even. What if `n` is odd? Then `n % 2` is `1` if `n` is positive or `-1` if `n` is negative. That handling of negative numbers is unfortunate in practice. Always be careful using `%` with potentially negative operands.

Consider this problem. You compute the position of the hour hand of a clock. An adjustment is applied, and you want to normalize to a number between 0 and 11. That is easy: `(position + adjustment) % 12`. But what if `adjustment` makes the position negative? Then you might get a negative number. So you have to introduce a branch, or use `((position + adjustment) % 12 + 12) % 12`. Either way, it is a hassle.

TIP: In this case, it is easier to use the `Math.floorMod` method: `Math.floorMod(position + adjustment, 12)` always yields a value between `0` and `11`.

Sadly, `floorMod` gives negative results for negative divisors, but that situation doesn't often occur in practice.

Java has increment and decrement operators:

```
n++; // Adds one to n
n--; // Subtracts one from n
```

As in other C-based languages, there is also a prefix form of these operators. Both `n++` and `++n` increment the variable `n`, but they have different values when they are used inside an expression. The first form yields the value before the increment, and the second the value after the increment. For example,

```
String arg = args[n++];
```

sets `arg` to `args[n]`, and *then* increments `n`. This made sense thirty years ago when compilers didn't do a good job optimizing code. Nowadays, there is no performance drawback in using two separate statements, and many programmers find the explicit form easier to read.

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. However, many modern processors use floating-point registers with more than 64 bit to add precision and reduce the risk of overflow in intermediate steps of a computation. Java allows these optimizations, since otherwise floating-point operations would be slower and less accurate. For the small set of users who care about this issue, there is a `strictfp` modifier. When added to a method, all floating-point operations in the method are strictly portable.

1.4.3 Mathematical Methods

There is no operator for raising numbers to a power. Instead, call the `Math.pow` method: `Math.pow(x, y)` yields x^y. To compute the square root of `x`, call `Math.sqrt(x)`.

These are *static* methods that don't operate on objects. Like with `static` constants, you prepend the name of the class in which they are declared.

Also useful are `Math.min` and `Math.max` for computing the minimum and maximum of two values.

In addition, the `Math` class provides trigonometric and logarithmic functions as well as the constants `Math.PI` and `Math.E`.

NOTE: The `Math` class provides several methods to make integer arithmetic safer. The mathematical operators quietly return wrong results when a computation overflows. For example, one billion times three (`1000000000 * 3`) evaluates to `-1294967296` because the largest `int` value is just over two billion. If you call `Math.multiplyExact(1000000000, 3)` instead, an exception is generated. You can catch that exception or let the program terminate rather than quietly continue with a wrong result. There are also methods `addExact`, `subtractExact`, `incrementExact`, `decrementExact`, `negateExact`, all with `int` and `long` parameters.

A few mathematical methods are in other classes. For example, there are methods `compareUnsigned`, `divideUnsigned`, and `remainderUnsigned` in the `Integer` and `Long` classes to work with unsigned values.

As discussed in the preceding section, some users require strictly reproducible floating-point computations even if they are less efficient. The `StrictMath` class provides strict implementations of mathematical methods.

1.4.4 Number Type Conversions

When an operator combines operands of different number types, the numbers are converted to a common type before they are combined. Conversion occurs in this order:

1. If either of the operands is of type `double`, the other one is converted to `double`.
2. If either of the operands is of type `float`, the other one is converted to `float`.
3. If either of the operands is of type `long`, the other one is converted to `long`.
4. Otherwise, both operands are converted to `int`.

For example, if you compute `3.14 + 42`, the second operand is converted to `42.0`, and then the sum is computed, yielding `45.14`.

If you compute `'J' + 1`, the `char` value `'J'` is converted to the `int` value `74`, and the result is the `int` value `75`. Read on to find out how to convert that value back to a `char`.

When you assign a value of a numeric type to a variable, or pass it as an argument to a method, and the types don't match, the value must be converted.

For example, in the assignment

```
double x = 42;
```

the value `42` is converted from `int` to `double`.

In Java, conversion is always legal if there is no loss of information:

- From `byte` to `short`, `int`, `long`, or `double`
- From `short` and `char` to `int`, `long`, or `double`
- From `int` to `long` or `double`

Conversion from an integer type to a floating-point type is always legal.

CAUTION: The following conversions are legal, but they may lose information:

- From `int` to `float`
- From `long` to `float` or `double`

For example, consider the assignment

```
float f = 123456789;
```

Because a `float` only has about seven significant digits, `f` is actually `1.23456792E8`.

To make a conversion that is not among these permitted ones, use a `cast` operator: the name of the target type in parentheses. For example,

```
double x = 3.75;
int n = (int) x;
```

In this case, the fractional part is discarded, and `n` is set to `3`.

If you want to round to the nearest integer instead, use the `Math.round` method. That method returns a `long`. If you know the answer fits into an `int`, call

```
int n = (int) Math.round(x);
```

In our example, where `x` is `3.75`, `n` is set to `4`.

To convert an integer type to another one with fewer bytes, also use a cast:

```
int n = 1;
char next = (char)('J' + n); // Converts 75 to 'K'
```

In such a cast, only the last bytes are retained.

```
int n = (int) 3000000000L; // Sets n to -1294967296
```

NOTE: If you worry that a cast can silently throw away important parts of a number, use the `Math.toIntExact` method instead. When it cannot convert a `long` to an `int`, an exception occurs.

1.4.5 Relational and Logical Operators

The `==` and `!=` operators test for equality. For example, `n != 0` is `true` when `n` is not zero.

There are also the usual `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal) operators.

You can combine expressions of type `boolean` with the `&&` (and), `||` (or), and `!` (not) operators. For example,

```
0 <= n && n < length
```

is `true` if `n` lies between zero (inclusive) and `length` (exclusive).

If the first condition is `false`, the second condition is not evaluated. This "short circuit" evaluation is useful when the second condition could cause an error. Consider the condition

```
n != 0 && s + (100 - s) / n < 50
```

If `n` is zero, then the second condition, which contains a division by `n`, is never evaluated, and no error occurs.

Short circuit evaluation is also used for "or" operations, but then the evaluation stops as soon as an operand is `true`. For example, the computation of

```
n == 0 || s + (100 - s) / n >= 50
```

yields `true` if `n` is zero, again without evaluating the second condition.

Finally, the *conditional* operator takes three operands: a condition and two values. The result is the first of the values if the condition is `true`, the second otherwise. For example,

```
time < 12 ? "am" : "pm"
```

yields the string `"am"` if `time < 12` and the string `"pm"` otherwise.

NOTE: There are *bitwise* operators `&` (and), `|` (or), and `^` (xor) that are related to the logical operators. They operate on the bit patterns of integers. For example, since `0xF` has binary digits `0...01111`, `n & 0xF` yields the lowest four bits in `n`, `n = n | 0xF` sets the lowest four bits to `1`, and `n = n ^ 0xF` flips them. The analog to the `!` operator is `~`, which flips all bits of its argument: `~0xF` is `1...10000`.

There are also operators which shift a bit pattern to left or right. For example, `0xF << 2` has binary digits `0...0111100`. There are two right shift operators: `>>` extends the sign bit into the top bits, and `>>>` fills the top bits with zero. If you do bit-fiddling in your programs, you know what that means. If not, you won't need these operators.

CAUTION: The right-hand side argument of the shift operators is reduced modulo 32 if the left hand side is an `int`, or modulo 64 if the left hand side is a `long`. For example, the value of `1 << 35` is the same as `1 << 3` or `8`.

TIP: The `&` (and) and `|` (or) operators, when applied to `boolean` values, force evaluation of both operands before combining the results. This usage is very uncommon. Provided that the right hand side doesn't have a side effect, they act just like `&&` and `||`, except they are less efficient. If you really need to force evaluation of the second operand, assign it to a `boolean` variable so that the flow of execution is plainly visible.

1.4.6 Big Numbers

If the precision of the primitive integer and floating-point types is not sufficient, you can turn to the `BigInteger` and `BigDecimal` classes in the `java.math` package. Objects of these classes represent 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.

The static `valueOf` method turns a `long` into a `BigInteger`:

```
BigInteger n = BigInteger.valueOf(876543210123456789L);
```

You can also construct a `BigInteger` from a string of digits:

```
BigInteger k = new BigInteger("9876543210123456789");
```

There are predefined constants `BigInteger.ZERO`, `BigInteger.ONE`, `BigInteger.TWO`, and `BigInteger.TEN`.

Java does not permit the use of operators with objects, so you must use method calls to work with big numbers.

```
BigInteger r = BigInteger.valueOf(5).multiply(n.add(k)); // r = 5 * (n + k)
```

In Section 1.2.2, "Floating-Point Types" (page 12), you saw that the result of the floating-point subtraction `2.0 - 1.1` is `0.8999999999999999`. The `BigDecimal` class can compute the result accurately.

The call `BigDecimal.valueOf(n, e)` returns a `BigDecimal` instance with value $n \times 10^{-e}$. The result of

```
BigDecimal.valueOf(2, 0).subtract(BigDecimal.valueOf(11, 1))
```

is exactly `0.9`.

1.5 Strings

A string is a sequence of characters. In Java, a string can contain any Unicode characters. For example, the string `"Java™"` or `"Java\u2122"` consists of the five characters `J`, `a`, `v`, `a`, and `™`. The last character is "U+2122 Trade Mark Sign."

1.5.1 Concatenation

Use the `+` operator to concatenate two strings. For example,

```
String location = "Java";
String greeting = "Hello " + location;
```

sets `greeting` to the string `"Hello Java"`. (Note the space at the end of the first operand.)

When you concatenate a string with another value, that value is converted to a string.

```
int age = 42;
String output = age + " years";
```

Now `output` is `"42 years"`.

CAUTION: If you mix concatenation and addition, then you may get unexpected results. For example,

```
"Next year, you will be " + age + 1 // Error
```

first concatenates `age` and then `1`. The result is `"Next year, you will be 421"`. In such cases, use parentheses:

```
"Next year, you will be " + (age + 1) // OK
```

To combine several strings, separated with a delimiter, use the `join` method:

```
String names = String.join(", ", "Peter", "Paul", "Mary");
    // Sets names to "Peter, Paul, Mary"
```

The first argument is the separator string, followed by the strings you want to join. There can be any number of them, or you can supply an array of strings. (Arrays are covered in Section 1.8, "Arrays and Array Lists," page 43.)

It is somewhat inefficient to concatenate a large number of strings if all you need is the final result. In that case, use a `StringBuilder` instead:

```
StringBuilder builder = new StringBuilder();
while (more strings) {
    builder.append(next string);
}
String result = builder.toString();
```

1.5.2 Substrings

To take strings apart, use the `substring` method. For example,

```
String greeting = "Hello, World!";
String location = greeting.substring(7, 12); // Sets location to "World"
```

The first argument of the `substring` method is the starting position of the substring to extract. Positions start at `0`.

The second argument is the first position that should not be included in the substring. In our example, position `12` of `greeting` is the `!`, which we do not want. It may seem curious to specify an unwanted position, but there is an advantage: the difference `12 - 7` is the length of the substring.

Sometimes, you want to extract all substrings from a string that are separated by a delimiter. The `split` method carries out that task, returning an array of substrings.

```
String names = "Peter, Paul, Mary";
String[] result = names.split(", ");
    // An array of three strings ["Peter", "Paul", "Mary"]
```

The separator can be any regular expression (see Chapter 9). For example, `input.split("\\s+")` splits `input` at white space.

1.5.3 String Comparison

To check whether two strings are equal, use the `equals` method. For example,

```
location.equals("World")
```

yields `true` if `location` is in fact the string `"World"`.

CAUTION: Never use the `==` operator to compare strings. The comparison

```
location == "World" // Don't do that!
```

returns `true` only if `location` and `"World"` are *the same object in memory*. In the virtual machine, there is only one instance of each literal string, so `"World" == "World"` will be `true`. But if `location` was computed, for example, as

```
String location = greeting.substring(7, 12);
```

then the result is placed into a separate `String` object, and the comparison `location == "World"` will return `false`!

Like any object variable, a `String` variable can be `null`, which indicates that the variable does not refer to any object at all, not even an empty string.

```
String middleName = null;
```

To test whether an object is `null`, you do use the `==` operator:

```
if (middleName == null) ...
```

Note that `null` is not the same as an empty string `""`. An empty string is a string of length zero, whereas `null` isn't any string at all.

CAUTION: Invoking any method on `null` causes a "null pointer exception." Like all exceptions, it terminates your program if you don't explicitly handle it.

TIP: When comparing a string against a literal string, it is a good idea to put the literal string first:

```
if ("World".equals(location)) ...
```

This test works correctly even when `location` is `null`.

To compare two strings without regard to case, use the `equalsIgnoreCase` method. For example,

```
"world".equalsIgnoreCase(location);
```

returns `true` if `location` is `"World"`, `"world"`, `"WORLD"`, and so on.

Sometimes, one needs to put strings in order. The `compareTo` method tells you whether one string comes before another in dictionary order. The call

```
first.compareTo(second)
```

returns a negative integer (not necessarily -1) if first comes before second, a positive integer (not necessarily 1) if first comes after second, and 0 if they are equal.

The strings are compared a character at a time, until one of them runs out of characters or a mismatch is found. For example, when comparing "**wor**d" and "**wor**ld", the first three characters match. Since d has a Unicode value that is less than that of l, "word" comes first. The call "word".compareTo("world") returns -8, the difference between the Unicode values of d and l.

This comparison can be unintuitive to humans because it depends on the Unicode values of characters. "blue/green" comes before "bluegreen" because / happens to have a lower Unicode value than g.

TIP: When sorting human-readable strings, use a Collator object that knows about language-specific sorting rules. See Chapter 13 for more information.

1.5.4 Converting Between Numbers and Strings

To turn an integer into a string, call the static Integer.toString method:

```
int n = 42;
String str = Integer.toString(n); // Sets str to "42"
```

A variant of this method has a second parameter, a radix (between 2 and 36):

```
String str2 = Integer.toString(n, 2); // Sets str2 to "101010"
```

NOTE: An even simpler way of converting an integer to a string is to concatenate with the empty string: "" + n. Some people find this ugly, and it is slightly less efficient.

Conversely, to convert a string containing an integer to the number, use the Integer.parseInt method:

```
String str = "101010";
int n = Integer.parseInt(str); // Sets n to 101010
```

You can also specify a radix:

```
int n2 = Integer.parseInt(str, 2); // Sets n2 to 42
```

For floating-point numbers, use Double.toString and Double.parseDouble:

```
String str = Double.toString(3.14); // Sets str to "3.14"
double x = Double.parseDouble(str); // Sets x to 3.14
```

1.5.5 The String API

As you might expect, the `String` class has a large number of methods. Some of the more useful ones are shown in Table 1-4.

Table 1-4 Useful String Methods

Method	Purpose

`boolean startsWith(String str)`
`boolean endsWith(String str)`
`boolean contains(CharSequence str)`
	Checks whether a string starts with, ends with, or contains a given string.

`int indexOf(String str)`
`int lastIndexOf(String str)`
`int indexOf(String str, int fromIndex)`
`int lastIndexOf(String str, int fromIndex)`
	Gets the position of the first or last occurrence of `str`, searching the entire string or the substring starting at `fromIndex`. Returns `-1` if no match is found.

`String replace(CharSequence oldString, CharSequence newString)`	Returns a string that is obtained by replacing all occurrences of `oldString` with `newString`.
`String toUpperCase()` `String toLowerCase()`	Returns a string consisting of all characters of the original string converted to upper- or lowercase.
`String trim()`	Returns a string obtained by removing all leading and trailing white space.

Note that in Java, the `String` class is *immutable*. That is, none of the `String` methods modify the string on which they operate. For example,

```
greeting.toUpperCase()
```

returns a *new* string `"HELLO, WORLD!"` without changing `greeting`.

Also note that some methods have parameters of type `CharSequence`. This is a common supertype of `String`, `StringBuilder`, and other sequences of characters.

For a detailed description of each method, turn to the online Java API documentation at `http://docs.oracle.com/javase/9/docs/api`. Type the class name into the search box and select the matching type (in this case, `java.lang.String`), as shown in Figure 1-4.

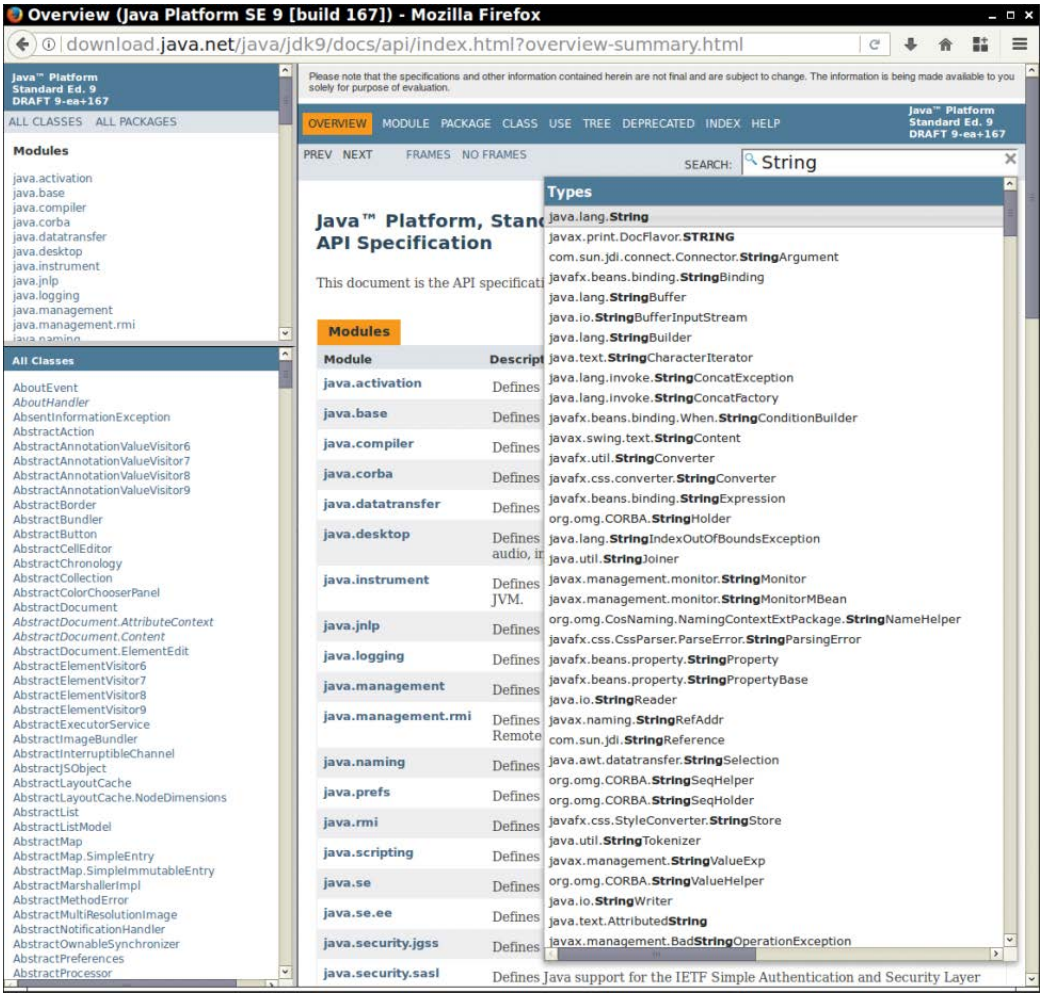

Figure 1-4 Searching the API Documentation

You then get a page that documents each method (Figure 1-5). If you happen to know the name of a method, you can type its name into the search box.

In this book, I do not present the API in minute detail since it is easier to browse the API documentation. If you are not always connected to the Internet, you can download and unzip the documentation for offline browsing.

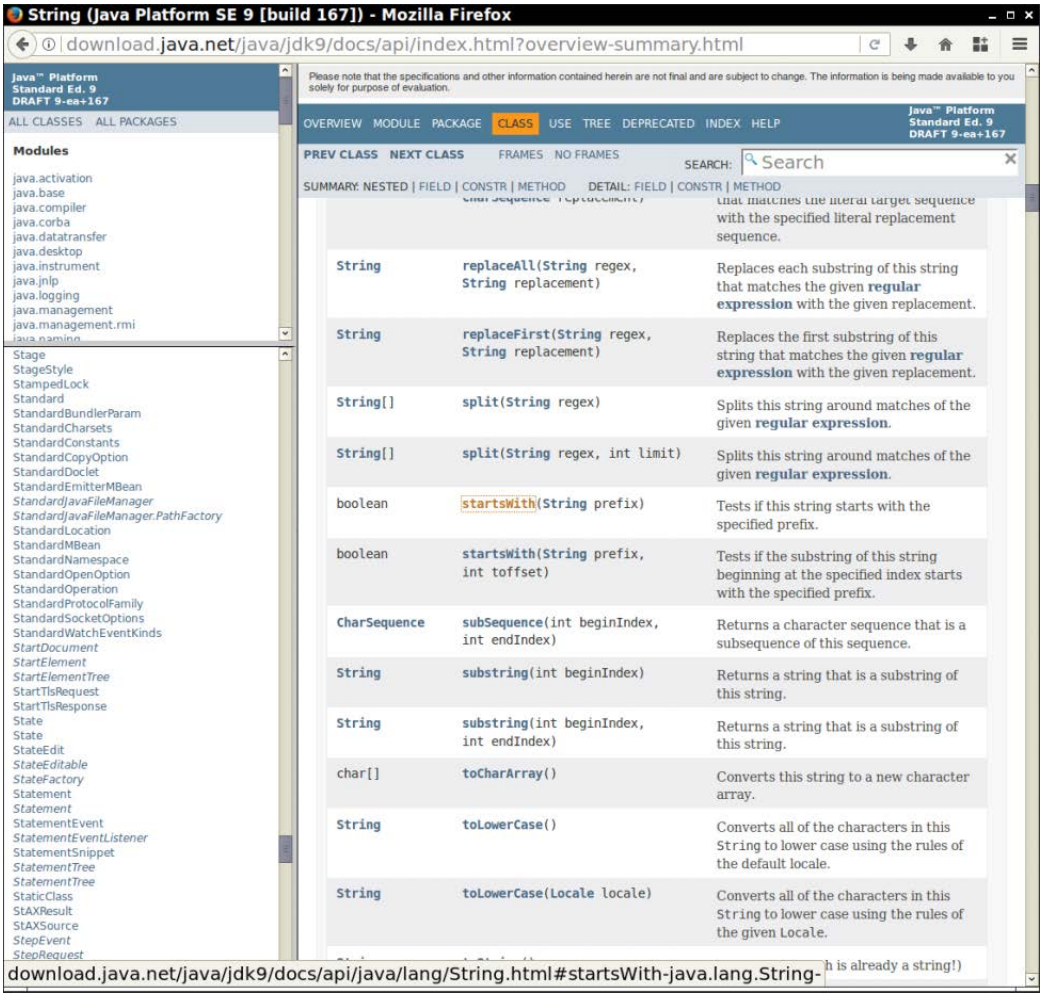

Figure 1–5 The `String` methods in the API Documentation

1.5.6 Code Points and Code Units

When Java was first created, it proudly embraced the Unicode standard that had been developed shortly before. The Unicode standard had been developed to solve a vexing issue of character encodings. Prior to Unicode, there were many incompatible character encodings. For English, there was near-universal agreement on the 7-bit ASCII standard that assigned codes between 0 and 127 to all English letters, the decimal digits, and many symbols. In Western Europe, ASCII was extended to an 8-bit code that contained accented characters such as ä and é. But in Russia, ASCII was extended to hold Cyrillic characters in the positions 128 to 255. In Japan, a variable-length encoding

was used to encode English and Japanese characters. Every other country did something similar. Exchanging files with different encodings was a major problem.

Unicode set out to fix all that by assigning each character in all of the writing systems ever devised a unique 16-bit code between 0 and 65535. In 1991, Unicode 1.0 was released, using slightly less than half of the available 65536 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. But then something awkward happened. There turned out to be many more characters than previously estimated—mostly Chinese ideographs. This pushed Unicode well beyond a 16-bit code.

Nowadays, Unicode requires 21 bits. Each valid Unicode value is called a *code point*. For example, the code point for the letter A is U+0041, and the mathematical symbol 𝕆 for the set of octonions (`http://math.ucr.edu/home/baez/octonions`) has code point U+1D546.

There is a variable-length backwards-compatible encoding, called UTF-16, that represents all "classic" Unicode characters with a single 16-bit value and the ones beyond U+FFFF as pairs of 16-bit values taken from a special region of the code space called "surrogate characters." In this encoding, the letter A is `\u0041` and 𝕆 is `\ud835\udd46`.

Java suffers from having been born at the time between the transition from 16 to 21 bits. Instead of having strings that are pristine sequences of Unicode characters or code points, Java strings are sequences of *code units*, the 16-bit quantities of the UTF-16 encoding.

If you don't need to worry about Chinese ideographs and are willing to throw special characters such as 𝕆 under the bus, then you can live with the fiction that a `String` is a sequence of Unicode characters. In that case, you can get the `i`th character as

```
char ch = str.charAt(i);
```

and the length of a string as

```
int length = str.length();
```

But if you want to handle strings properly, you have to work harder.

To get the `i`th Unicode code point, call

```
int codePoint = str.codePointAt(str.offsetByCodePoints(0, i));
```

The total number of code points is

```
int length = str.codePointCount(0, str.length());
```

This loop extracts the code points sequentially:

```
int i = 0;
while (i < s.length()) {
    int j = s.offsetByCodePoints(i, 1);
    String codePoint = str.substring(i, j);
    ...
    i = j;
}
```

Alternatively, you can use the `codePoints` method that yields a *stream* of `int` values, one for each code point. We will discuss streams in Chapter 8. You can turn the stream into an array like this:

```
int[] codePoints = str.codePoints().toArray();
```

NOTE: In the past, strings were always internally represented in the UTF-16 encoding, as arrays of `char` values. Nowadays, `String` objects use a `byte` array of ISO-8859-1 characters when possible. A future version of Java may switch to using UTF-8 internally.

1.6 Input and Output

To make our sample programs more interesting, they should be able to interact with the user. In the following sections, you will see how to read terminal input and how to produce formatted output.

1.6.1 Reading Input

When you call `System.out.println`, output is sent to the "standard output stream" and shows up in a terminal window. Reading from the "standard input stream" isn't quite as simple because the corresponding `System.in` object only has methods to read individual bytes. To read strings and numbers, construct a `Scanner` that is attached to `System.in`:

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

The `nextLine` method reads a line of input.

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

Here, it makes sense to 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.println("How old are you?");
int age = in.nextInt();
```

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

You can use the `hasNextLine`, `hasNext`, `hasNextInt`, and `hasNextDouble` methods to check that there is another line, word, integer, or floating-point number available.

```
if (in.hasNextInt()) {
    int age = in.nextInt();
    ...
}
```

The `Scanner` class is located in the `java.util` package. In order to use the class, add the line

```
import java.util.Scanner
```

to the top of your program file.

TIP: To read a password, you do not want to use the `Scanner` class since the input is visible in the terminal. Instead, use the `Console` class:

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

The password is returned in an array of characters. This is marginally more secure than storing the password in a `String` because you can overwrite the array when you are done.

TIP: If you want to read input from a file or write output to a file, you can use the redirection syntax of your shell:

```
java mypackage.MainClass < input.txt > output.txt
```

Now `System.in` reads from `input.txt` and `System.out` writes to `output.txt`. You will see in Chapter 9 how to carry out more general file input and output.

1.6.2 Formatted Output

You have already seen the `println` method of the `System.out` object for writing a line of output. There is also a `print` method that does not start a new line. That method is often used for input prompts:

```
System.out.print("Your age: "); // Not println
int age = in.nextInt();
```

Then the cursor rests after the prompt instead of the next line.

When you print a fractional number with `print` or `println`, all of its digits except trailing zeroes will be displayed. For example,

```
System.out.print(1000.0 / 3.0);
```

prints

```
333.3333333333333
```

That is a problem if you want to display, for example, dollars and cents. To limit the number of digits, use the `printf` method:

```
System.out.printf("%8.2f", 1000.0 / 3.0);
```

The *format string* `"%8.2f"` indicates that a floating-point number is printed with a *field width* of 8 and 2 digits of *precision*. That is, the printout contains two leading spaces and six characters:

```
  333.33
```

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

```
System.out.printf("Hello, %s. Next year, you'll be %d.\n", 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 1-5 shows all conversion characters.

Table 1–5 Conversion Characters for Formatted Output

Conversion Character	Purpose	Example
`d`	Decimal integer	`159`
`x` or `X`	Hexadecimal integer	`9f` or `9F`
`o`	Octal integer	`237`
`f`	Fixed floating-point	`15.9`
`e` or `E`	Exponential floating-point	`1.59e+01` or `1.59E+01`
`g` or `G`	General floating-point: `e`/`E` if the exponent is greater than the precision or < −4, `f`/`F` otherwise	`15.9000` at the default precision of 6, `2e+01` at precision 1
`a` or `A`	Hexadecimal floating-point	`0x1.fccdp3` or `0X1.FCCDP3`
`s` or `S`	String	`Java` or `JAVA`
`c` or `C`	Character	`j` or `J`

(Continues)

Table 1-5 Conversion Characters for Formatted Output *(Continued)*

Conversion Character	Purpose	Example
`b` or `B`	`boolean`	`false` or `FALSE`
`h` or `H`	Hash code (see Chapter 4)	`42628b2` or `42628B2`
`t` or `T`	Date and time (obsolete; see Chapter 12 instead)	—
`%`	The percent symbol	`%`
`n`	The platform-dependent line separator	—

In addition, you can specify flags to control the appearance of the formatted output. Table 1-6 shows all flags. For example, the comma flag adds grouping separators, and `+` yields a sign for positive numbers. The statement

```
System.out.printf("%,+.2f", 100000.0 / 3.0);
```

prints

```
+33,333.33
```

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

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

Table 1-6 Flags for Formatted Output

Flag	Purpose	Example
`+`	Prints sign for positive and negative numbers	`+3333.33`
space	Adds a space before positive numbers	`_3333.33`
`-`	Left-justifies field	`3333.33___`
`0`	Adds leading zeroes	`003333.33`
`(`	Encloses negative values in parentheses	`(3333.33)`
`,`	Uses group separators	`3,333.33`
`#` (for `f` format)	Always includes a decimal point	`3333.`
`#` (for `x` or `o` format)	Adds `0x` or `0` prefix	`0xcafe`

(Continues)

Table 1-6 Flags for Formatted Output *(Continued)*

Flag	Purpose	Example
$	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`

1.7 Control Flow

In the following sections, you will see how to implement branches and loops. The Java syntax for control flow statements is very similar to that of other commonly used languages, in particular C/C++ and JavaScript.

1.7.1 Branches

The `if` statement has a condition in parentheses, followed by either one statement or a group of statements enclosed in braces.

```
if (count > 0) {
    double average = sum / count;
    System.out.println(average);
}
```

You can have an `else` branch that runs if the condition is not fulfilled.

```
if (count > 0) {
    double average = sum / count;
    System.out.println(average);
} else {
    System.out.println(0);
}
```

The statement in the `else` branch may be another `if` statement:

```
if (count > 0) {
    double average = sum / count;
    System.out.println(average);
} else if (count == 0) {
    System.out.println(0);
} else {
    System.out.println("Huh?");
}
```

When you need to test an expression against a finite number of constant values, use the `switch` statement.

```
switch (count) {
    case 0:
        output = "None";
        break;
    case 1:
        output = "One";
        break;
    case 2:
    case 3:
    case 4:
    case 5:
        output = Integer.toString(count);
        break;
    default:
        output = "Many";
        break;
}
```

Execution starts at the matching `case` label or, if there is no match, at the `default` label (if it is present). All statements are executed until a `break` or the end of the `switch` statement is reached.

CAUTION: It is a common error to forget a `break` at the end of an alternative. Then execution "falls through" to the next alternative. You can direct the compiler to be on the lookout for such bugs with a command-line option:

```
javac -Xlint:fallthrough mypackage/MainClass.java
```

With this option, the compiler will issue a warning message whenever an alternative does not end with a `break` or `return` statement.

If you actually want to use the fallthrough behavior, tag the surrounding method with the *annotation* `@SuppressWarnings("fallthrough")`. Then no warnings will be generated for that method. (An annotation supplies information to the compiler or another tool. You will learn all about annotations in Chapter 11.)

In the preceding example, the `case` labels were integers. You can use values of any of the following types:

- A constant expression of type `char`, `byte`, `short`, or `int` (or their corresponding wrapper classes `Character`, `Byte`, `Short`, and `Integer` that will be introduced in Section 1.8.3, "Array Lists," page 45)

- A string literal
- A value of an enumeration (see Chapter 4)

1.7.2 Loops

The `while` loop keeps executing its body while more work needs to be done, as determined by a condition.

For example, consider the task of summing up numbers until the sum has reached a target. For the source of numbers, we will use a random number generator, provided by the `Random` class in the `java.util` package.

```
Random generator = new Random();
```

This call gets a random integer between `0` and `9`:

```
int next = generator.nextInt(10);
```

Here is the loop for forming the sum:

```
while (sum < target) {
    int next = generator.nextInt(10);
    sum += next;
    count++;
}
```

This is a typical use of a `while` loop. While the sum is less than the target, the loop keeps executing.

Sometimes, you need to execute the loop body before you can evaluate the condition. Suppose you want to find out how long it takes to get a particular value. Before you can test that condition, you need to enter the loop and get the value. In this case, use a `do/while` loop:

```
int next;
do {
    next = generator.nextInt(10);
    count++;
} while (next != target);
```

The loop body is entered, and `next` is set. Then the condition is evaluated. As long as it is fulfilled, the loop body is repeated.

In the preceding examples, the number of loop iterations was not known. However, in many loops that occur in practice, the number of iterations is fixed. In those situations, it is best to use the `for` loop.

This loop computes the sum of a fixed number of random values:

```
for (int i = 1; i <= 20; i++) {
    int next = generator.nextInt(10);
    sum += next;
}
```

This loop runs 20 times, with `i` set to 1, 2, ..., 20 in each loop iteration.

You can rewrite any `for` loop as a `while` loop. The loop above is equivalent to

```
int i = 1;
while (i <= 20) {
    int next = generator.nextInt(10);
    sum += next;
    i++;
}
```

However, with the `while` loop, the initialization, test, and update of the variable `i` are scattered in different places. With the `for` loop, they stay neatly together.

The initialization, test, and update can take on arbitrary forms. For example, you can double a value while it is less than the target:

```
for (int i = 1; i < target; i *= 2) {
    System.out.println(i);
}
```

Instead of declaring a variable in the header of the `for` loop, you can initialize an existing variable:

```
for (i = 1; i <= target; i++) // Uses existing variable i
```

You can declare or initialize multiple variables and provide multiple updates, separated by commas. For example,

```
for (int i = 0, j = n - 1; i < j; i++, j--)
```

If no initialization or update is required, leave them blank. If you omit the condition, it is deemed to always be `true`.

```
for (;;) // An infinite loop
```

You will see in the next section how you can break out of such a loop.

1.7.3 Breaking and Continuing

If you want to exit a loop in the middle, you can use the `break` statement. For example, suppose you want to process words until the user enters the letter Q. Here is a solution that uses a `boolean` variable to control the loop:

```
boolean done = false;
while (!done) {
    String input = in.next();
    if ("Q".equals(input)) {
        done = true;
    } else {
        Process input
    }
}
```

This loop carries out the same task with a `break` statement:

```
while (true) {
    String input = in.next();
    if (input.equals("Q")) break; // Exits loop
    Process input
}
// break jumps here
```

When the `break` statement is reached, the loop is exited immediately.

The `continue` statement is similar to `break`, but instead of jumping to the end of the loop, it jumps to the end of the current loop iteration. You might use it to skip unwanted inputs like this:

```
while (in.hasNextInt()) {
    int input = in.nextInt();
    if (input < 0) continue; // Jumps to test of in.hasNextInt()
    Process input
}
```

In a `for` loop, the `continue` statement jumps to the next update statement:

```
for (int i = 1; i <= target; i++) {
    int input = in.nextInt();
    if (n < 0) continue; // Jumps to i++
    Process input
}
```

The `break` statement only breaks out of the immediately enclosing loop or `switch`. If you want to jump to the end of another enclosing statement, use a *labeled* `break` statement. Label the statement that should be exited, and provide the label with the `break` like this:

```
outer:
while (...) {
    ...
    while (...) {
        ...
        if (...) break outer;
       ...
     }
     ...
}
// Labeled break jumps here
```

The label can be any name.

CAUTION: You label the top of the statement, but the `break` statement jumps to the *end*.

A regular `break` can only be used to exit a loop or `switch`, but a labeled `break` can transfer control to the end of any statement, even a block statement:

```
exit: {
    ...
    if (...) break exit;
    ...
}
// Labeled break jumps here
```

There is also a labeled `continue` statement that jumps to the next iteration of a labeled loop.

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, I never use `break` or `continue`.

1.7.4 Local Variable Scope

Now that you have seen examples of nested blocks, it is a good idea to go over the rules for variable scope. A *local variable* is any variable that is declared in a method, including the method's parameter variables. The *scope* of a variable is the part of the program where you can access the variable. The scope of a local variable extends from the point where it is declared to the end of the enclosing block.

```
while (...) {
    System.out.println(...);
    String input = in.next(); // Scope of input starts here
    ...
    // Scope of input ends here
}
```

In other words, a new copy of `input` is created for each loop iteration, and the variable does not exist outside the loop.

The scope of a parameter variable is the entire method.

```
public static void main(String[] args) { // Scope of args starts here
    ...
    // Scope of args ends here
}
```

Here is a situation where you need to understand scope rules. This loop counts how many tries it takes to get a particular random digit:

```
int next;
do {
    next = generator.nextInt(10);
    count++;
} while (next != target);
```

The variable `next` had to be declared outside the loop so it is available in the condition. Had it been declared inside the loop, its scope would only reach to the end of the loop body.

When you declare a variable in a `for` loop, its scope extends to the end of the loop, including the test and update statements.

```
for (int i = 0; i < n; i++) { // i is in scope for the test and update
    ...
}
// i not defined here
```

If you need the value of `i` after the loop, declare the variable outside:

```
int i;
for (i = 0; !found && i < n; i++) {
    ...
}
// i still available
```

In Java, you cannot have local variables with the same name in overlapping scopes.

```
int i = 0;
while (...) {
    String i = in.next(); // Error to declare another variable i
    ...
}
```

However, if the scopes do not overlap, you can reuse the same variable name:

```
for (int i = 0; i < n / 2; i++) { ... }
for (int i = n / 2; i < n; i++) { ... } // OK to redefine i
```

1.8 Arrays and Array Lists

Arrays are a fundamental programming construct for collecting multiple items of the same type. Java has array types built into the language, and it also supplies an `ArrayList` class for arrays that grow and shrink on demand. The `ArrayList` class is a part of a larger collections framework that is covered in Chapter 7.

1.8.1 Working with Arrays

For every type, there is a corresponding array type. An array of integers has type `int[]`, an array of `String` objects has type `String[]`, and so on. Here is a variable that can hold an array of strings:

```
String[] names;
```

The variable isn't yet initialized. Let's initialize it with a new array. For that, we need the `new` operator:

```
names = new String[100];
```

Of course, you can combine these two statements:

```
String[] names = new String[100];
```

Now `names` refers to an array with 100 elements, which you can access as `names[0]` ... `names[99]`.

CAUTION: If you try to access an element that does not exist, such as `names[-1]` or `names[100]`, an `ArrayIndexOutOfBoundsException` occurs.

The length of an array can be obtained as *array*`.length`. For example, this loop fills the array with empty strings:

```
for (int i = 0; i < names.length; i++) {
    names[i] = "";
}
```

NOTE: It is legal to use the C syntax for declaring an array variable, with the `[]` following the variable name:

```
int numbers[];
```

However, this syntax is unfortunate since it intertwines the name `numbers` and the type `int[]`. Few Java programmers use it.

1.8.2 Array Construction

When you construct an array with the `new` operator, it is filled with a default value.

- Arrays of numeric type (including `char`) are filled with zeroes.
- Arrays of `boolean` are filled with `false`.
- Arrays of objects are filled with `null` references.

CAUTION: Whenever you construct an array of objects, you need to fill it with objects. Consider this declaration:

```
BigInteger[] numbers = new BigInteger[100];
```

At this point, you do not have any `BigInteger` objects yet, just an array of 100 `null` references. You need to replace them with references to `BigInteger` objects:

```
for (int i = 0; i < 100; i++)
    numbers[i] = BigInteger.valueOf(i);
```

You can fill an array with values by writing a loop, as you saw in the preceding section. However, sometimes you know the values that you want, and you can just list them inside braces:

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

You don't use the `new` operator, and you don't specify the array length. A trailing comma is allowed, which can be convenient for an array to which you keep adding values over time:

```
String[] authors = {
    "James Gosling",
    "Bill Joy",
    "Guy Steele",
    // Add more names here and put a comma after each name
};
```

Use a similar initialization syntax if you don't want to give the array a name—for example, to assign it to an existing array variable:

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

NOTE: It is legal to have arrays of length 0. You can construct such an array as `new int[0]` or `new int[] {}`. For example, if a method returns an array of matches, and there weren't any for a particular input, return an array of length 0. Note that this is not the same as `null`: If `a` is an array of length 0, then `a.length` is `0`; if `a` is `null`, then `a.length` causes a `NullPointerException`.

1.8.3 Array Lists

When you construct an array, you need to know its length. Once constructed, the length can never change. That is inconvenient in many practical applications. A remedy is to use the `ArrayList` class in the `java.util` package. An `ArrayList` object manages an array internally. When that array becomes too small or is insufficiently utilized, another internal array is automatically created, and the elements are moved into it. This process is invisible to the programmer using the array list.

The syntax for arrays and array lists is completely different. Arrays use a special syntax—the `[]` operator for accessing elements, the *Type*`[]` syntax for array types, and the `new` *Type*`[`*n*`]` syntax for constructing arrays. In contrast, array lists are classes, and you use the normal syntax for constructing instances and invoking methods.

However, unlike the classes that you have seen so far, the `ArrayList` class is a *generic class*—a class with a type parameter. Chapter 6 covers generic classes in detail.

To declare an array list variable, you use the syntax for generic classes and specify the type in angle brackets:

```
ArrayList<String> friends;
```

As with arrays, this only declares the variable. You now need to construct an array list:

```
friends = new ArrayList<>();
   // or new ArrayList<String>()
```

Note the empty `<>`. The compiler infers the type parameter from the type of the variable. (This shortcut is called the *diamond syntax* because the empty angle brackets have the shape of a diamond.)

There are no construction arguments in this call, but it is still necessary to supply the `()` at the end.

The result is an array list of size 0. You can add elements to the end with the `add` method:

```
friends.add("Peter");
friends.add("Paul");
```

Unfortunately, there is no initializer syntax for array lists. The best you can do is construct an array list like this:

```
ArrayList<String> friends = new ArrayList<>(List.of("Peter", "Paul"));
```

The `List.of` method yields an unmodifiable list of the given elements which you then use to construct an `ArrayList`.

You can add and remove elements anywhere in the `ArrayList`.

```
friends.remove(1);
friends.add(0, "Paul"); // Adds before index 0
```

To access elements, use method calls, not the `[]` syntax. The `get` method reads an element, and the `set` method replaces an element with another:

```
String first = friends.get(0);
friends.set(1, "Mary");
```

The `size` method yields the current size of the list. Use the following loop to traverse all elements:

```
for (int i = 0; i < friends.size(); i++) {
    System.out.println(friends.get(i));
}
```

1.8.4 Wrapper Classes for Primitive Types

There is one unfortunate limitation of generic classes: You cannot use primitive types as type parameters. For example, an `ArrayList<int>` is illegal. The remedy is to use a *wrapper class*. For each primitive type, there is a corresponding wrapper class: `Integer`, `Byte`, `Short`, `Long`, `Character`, `Float`, `Double`, and `Boolean`. To collect integers, use an `ArrayList<Integer>`:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(42);
int first = numbers.get(0);
```

Conversion between primitive types and their corresponding wrapper types is automatic. In the call to `add`, an `Integer` object holding the value `42` was automatically constructed in a process called *autoboxing*.

In the last line of the code segment, the call to `get` returned an `Integer` object. Before assigning to the `int` variable, the object was *unboxed* to yield the `int` value inside.

CAUTION: Conversion between primitive types and wrappers is almost completely transparent to programmers, with one exception. The `==` and `!=` operators compare object references, not the contents of objects. A condition `if (numbers.get(i) == numbers.get(j))` does not test whether the numbers at index `i` and `j` are the same. Just like with strings, you need to remember to call the `equals` method with wrapper objects.

1.8.5 The Enhanced for Loop

Very often, you want to visit all elements of an array. For example, here is how you compute the sum of all elements in an array of numbers:

```
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
```

As this loop is so common, there is a convenient shortcut, called the *enhanced* `for` loop:

```
int sum = 0;
for (int n : numbers) {
    sum += n;
}
```

The loop variable of the enhanced `for` loop traverses the elements of the array, not the index values. The variable `n` is assigned to `numbers[0]`, `numbers[1]`, and so on.

You can also use the enhanced `for` loop with array lists. If `friends` is an array list of strings, you can print them all with the loop

```
for (String name : friends) {
    System.out.println(name);
}
```

1.8.6 Copying Arrays and Array Lists

You can copy one array variable into another, but then both variables will refer to the same array, as shown in Figure 1-6.

```
int[] numbers = primes;
numbers[5] = 42; // Now primes[5] is also 42
```

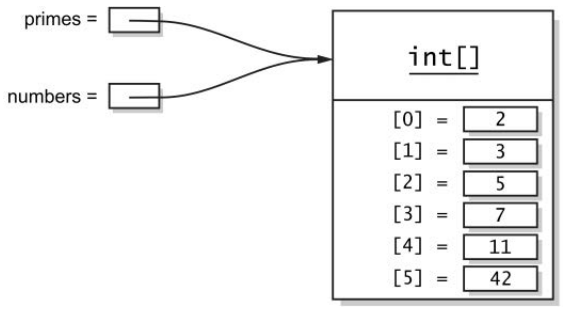

Figure 1-6 Two variables referencing the same array

If you don't want this sharing, you need to make a copy of the array. Use the static `Arrays.copyOf` method.

```
int[] copiedPrimes = Arrays.copyOf(primes, primes.length);
```

This method constructs a new array of the desired length and copies the elements of the original array into it.

Array list references work the same way:

```
ArrayList<String> people = friends;
people.set(0, "Mary"); // Now friends.get(0) is also "Mary"
```

To copy an array list, construct a new array list from the existing one:

```
ArrayList<String> copiedFriends = new ArrayList<>(friends);
```

That constructor can also be used to copy an array into an array list. Wrap the array into an immutable list, using the `List.of` method, and then construct an `ArrayList`:

```
String[] names = ...;
ArrayList<String> friends = new ArrayList<>(List.of(names));
```

You can also copy an array list into an array. For depressing reasons of backward compatibility that I will explain in Chapter 6, you must supply an array of the correct type.

```
String[] names = friends.toArray(new String[0]);
```

NOTE: There is no easy way to convert between primitive type arrays and the corresponding array lists of wrapper classes. For example, to convert between an `int[]` and an `ArrayList<Integer>`, you need an explicit loop or an `IntStream` (see Chapter 8).

1.8.7 Array Algorithms

The `Arrays` and `Collections` classes provide implementations of common algorithms for arrays and array lists. Here is how to fill an array or an array list:

```
Arrays.fill(numbers, 0); // int[] array
Collections.fill(friends, ""); // ArrayList<String>
```

To sort an array or array list, use the `sort` method:

```
Arrays.sort(names);
Collections.sort(friends);
```

NOTE: For arrays (but not array lists), you can use the `parallelSort` method that distributes the work over multiple processors if the array is large.

The `Arrays.toString` method yields a string representation of an array. This is particularly useful to print an array for debugging.

```
System.out.println(Arrays.toString(primes));
    // Prints [2, 3, 5, 7, 11, 13]
```

Array lists have a `toString` method that yields the same representation.

```
String elements = friends.toString();
    // Sets elements to "[Peter, Paul, Mary]"
```

For printing, you don't even need to call it—the `println` method takes care of that.

```
System.out.println(friends);
    // Calls friends.toString() and prints the result
```

There are a couple of useful algorithms for array lists that have no counterpart for arrays.

```
Collections.reverse(names); // Reverses the elements
Collections.shuffle(names); // Randomly shuffles the elements
```

1.8.8 Command-Line Arguments

As you have already seen, the `main` method of every Java program has a parameter that is a string array:

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

When a program is executed, this parameter is set to the arguments specified on the command line.

For example, consider this program:

```
public class Greeting {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            String arg = args[i];
            if (arg.equals("-h")) arg = "Hello";
            else if (arg.equals("-g")) arg = "Goodbye";
            System.out.println(arg);
        }
    }
}
```

If the program is called as

```
java Greeting -g cruel world
```

then `args[0]` is `"-g"`, `args[1]` is `"cruel"`, and `args[2]` is `"world"`.

Note that neither `"java"` nor `"Greeting"` are passed to the `main` method.

1.8.9 Multidimensional Arrays

Java does not have true multidimensional arrays. They are implemented as arrays of arrays. For example, here is how you declare and implement a two-dimensional array of integers:

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

Technically, this is a one-dimensional array of `int[]` arrays—see Figure 1-7.

To access an element, use two bracket pairs:

```
int element = square[1][2]; // Sets element to 11
```

The first index selects the row array `square[1]`. The second index picks the element from that row.

You can even swap rows:

```
int[] temp = square[0];
square[0] = square[1];
square[1] = temp;
```

If you do not provide an initial value, you must use the `new` operator and specify the number of rows and columns.

```
int[][] square = new int[4][4]; // First rows, then columns
```

Behind the scenes, an array of rows is filled with an array for each row.

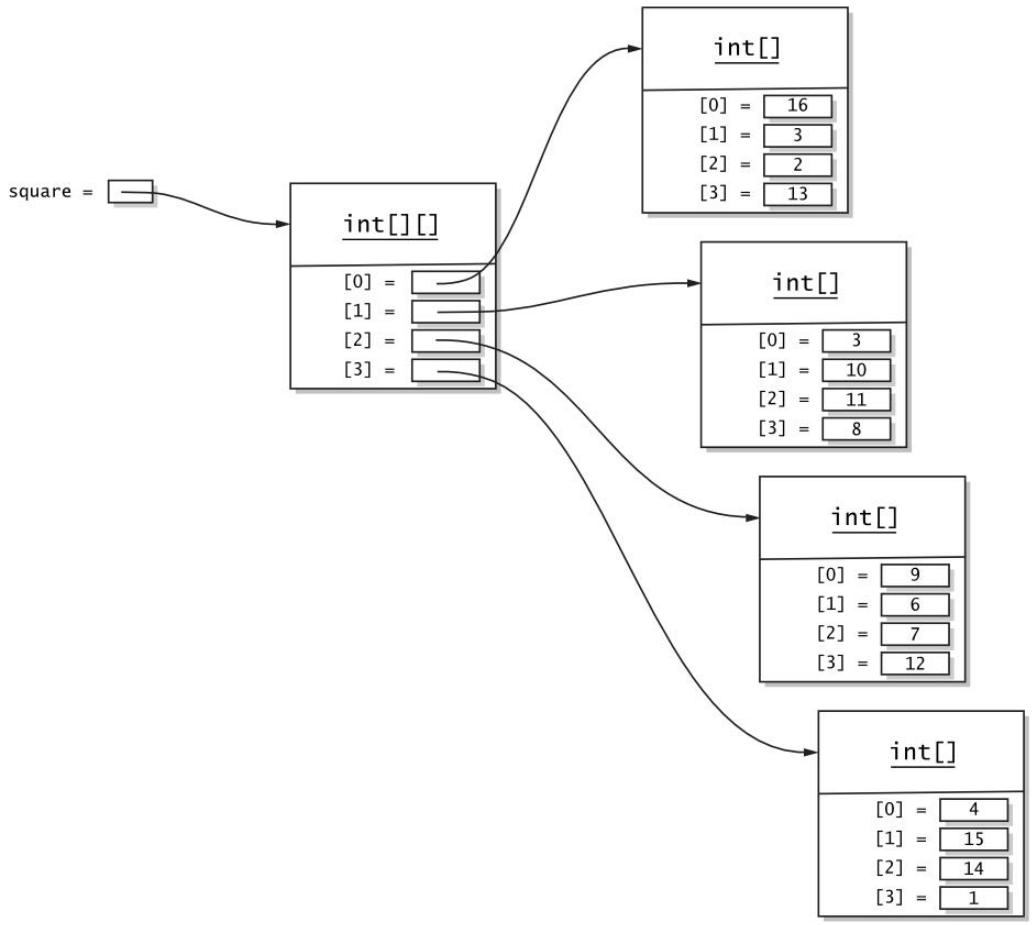

Figure 1-7 A two-dimensional array

There is no requirement that the row arrays have equal length. For example, you can store the Pascal triangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

First construct an array of `n` rows:

```
int[][] triangle = new int[n][];
```

Then construct each row in a loop and fill it.

```
for (int i = 0; i < n; i++) {
    triangle[i] = new int[i + 1];
    triangle[i][0] = 1;
    triangle[i][i] = 1;
    for (int j = 1; j < i; j++) {
        triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
    }
}
```

To traverse a two-dimensional array, you need two loops, one for the rows and one for the columns:

```
for (int r = 0; r < triangle.length; r++) {
    for (int c = 0; c < triangle[r].length; c++) {
        System.out.printf("%4d", triangle[r][c]);
    }
    System.out.println();
}
```

You can also use two enhanced `for` loops:

```
for (int[] row : triangle) {
    for (int element : row) {
       System.out.printf("%4d", element);
    }
    System.out.println();
}
```

These loops work for square arrays as well as arrays with varying row lengths.

TIP: To print out a list of the elements of a two-dimensional array for debugging, call

```
System.out.println(Arrays.deepToString(triangle));
    // Prints [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], ...]
```

NOTE: There are no two-dimensional array lists, but you can declare a variable of type `ArrayList<ArrayList<Integer>>` and build up the rows yourself.

1.9 Functional Decomposition

If your `main` method gets too long, you can decompose your program into multiple classes, as you will see in Chapter 2. However, for simple programs, you can place your program's code into separate methods in the same class. For reasons that will become clear in Chapter 2, these methods must be declared with the `static` modifier, just as the `main` method itself.

1.9.1 Declaring and Calling Static Methods

When you declare a method, provide the type of the return value (or `void` if the method doesn't return anything), the method name, and the types and names of the parameters in the *method header*. Then provide the implementation in the *method body*. Use a `return` statement to return the result.

```
public static double average(double x, double y) {
    double sum = x + y;
    return sum / 2;
}
```

Place the method in the same class as the `main` method. It doesn't matter if it's above or below `main`. Then, call it like this:

```
public static void main(String[] args) {
    double a = ...;
    double b = ...;
    double result = average(a, b);
    ...
}
```

1.9.2 Array Parameters and Return Values

You can pass arrays into methods. The method simply receives a reference to the array, through which it can modify it. This method swaps two elements in an array:

```
public static void swap(int[] values, int i, int j) {
    int temp = values[i];
    values[i] = values[j];
    values[j] = temp;
}
```

Methods can return arrays. This method returns an array consisting of the first and last values of a given array (which is not modified):

```
public static int[] firstLast(int[] values) {
    if (values.length == 0) return new int[0];
    else return new int[] { values[0], values[values.length - 1] };
}
```

1.9.3 Variable Arguments

Some methods allow the caller to supply a variable number of arguments. You have already seen such a method: `printf`. For example, the calls

```
System.out.printf("%d", n);
```

and

```
System.out.printf("%d %s", n, "widgets");
```

both call the same method, even though one call has two arguments and the other has three.

Let us define an `average` method that works the same way, so we can call `average` with as many arguments as we like, for example, `average(3, 4.5, -5, 0)`. Declare a "varargs" parameter with ... after the type:

```
public static double average(double... values)
```

The parameter is actually an array of type `double`. When the method is called, an array is created and filled with the arguments. In the method body, you use it as you would any other array.

```
public static double average(double... values) {
    double sum = 0;
    for (double v : values) sum += v;
    return values.length == 0 ? 0 : sum / values.length;
}
```

Now you can call

```
double avg = average(3, 4.5, -5, 0);
```

If you already have the arguments in an array, you don't have to unpack them. You can pass the array instead of the list of arguments:

```
double[] scores = { 3, 4.5, -5, 0 };
double avg = average(scores);
```

The variable parameter must be the *last* parameter of the method, but you can have other parameters before it. For example, this method ensures that there is at least one argument:

```
public static double max(double first, double... rest) {
    double result = first;
    for (double v : rest) result = Math.max(v, result);
    return result;
}
```

Exercises

1. Write a program that reads an integer and prints it in binary, octal, and hexadecimal. Print the reciprocal as a hexadecimal floating-point number.
2. Write a program that reads an integer angle (which may be positive or negative) and normalizes it to a value between 0 and 359 degrees. Try it first with the `%` operator, then with `floorMod`.

3. Using only the conditional operator, write a program that reads three integers and prints the largest. Repeat with `Math.max`.
4. Write a program that prints the smallest and largest positive `double` values. Hint: Look up `Math.nextUp` in the Java API.
5. What happens when you cast a `double` to an `int` that is larger than the largest possible `int` value? Try it out.
6. Write a program that computes the factorial $n! = 1 \times 2 \times \ldots \times n$, using `BigInteger`. Compute the factorial of 1000.
7. Write a program that reads in two integers between 0 and 4294967295, stores them in `int` variables, and computes and displays their unsigned sum, difference, product, quotient, and remainder. Do not convert them to `long` values.
8. Write a program that reads a string and prints all of its nonempty substrings.
9. Section 1.5.3, "String Comparison" (page 25) has an example of two strings `s` and `t` so that `s.equals(t)` but `s != t`. Come up with a different example that doesn't use `substring`).
10. Write a program that produces a random string of letters and digits by generating a random `long` value and printing it in base 36.
11. Write a program that reads a line of text and prints all characters that are not ASCII, together with their Unicode values.
12. The Java Development Kit includes a file `src.zip` with the source code of the Java library. Unzip and, with your favorite text search tool, find usages of the labeled `break` and `continue` sequences. Take one and rewrite it without a labeled statement.
13. Write a program that prints a lottery combination, picking six distinct numbers between 1 and 49. To pick six distinct numbers, start with an array list filled with 1…49. Pick a random index and remove the element. Repeat six times. Print the result in sorted order.
14. Write a program that reads a two-dimensional array of integers and determines whether it is a magic square (that is, whether the sum of all rows, all columns, and the diagonals is the same). Accept lines of input that you break up into individual integers, and stop when the user enters a blank line. For example, with the input

```
16 3 2 13
5 10 11 8
9 6 7 12
4 15 14 1
(Blank line)
```

your program should respond affirmatively.

15. Write a program that stores Pascal's triangle up to a given n in an `ArrayList<ArrayList<Integer>>`.
16. Improve the `average` method so that it is called with at least one parameter.