Chapter 3

Introduction to Parameters and Objects

Introduction

Chapter 2 introduced techniques for managing complexity, including the use of class constants, which make programs more flexible. This chapter explores a more powerful technique for obtaining such flexibility. Here, you will learn how to use parameters to create methods that solve not just single tasks, but whole families of tasks. Creating such methods requires you to generalize, or look beyond a specific task to find the more general category of task that it exemplifies. The ability to generalize is one of the most important qualities of a good software engineer, and the generalization technique you will study in this chapter is one of the most powerful techniques programmers use. After exploring parameters, we'll discuss some other issues associated with methods, such as the ability of a method to return a value.

This chapter then introduces the idea of objects and how to use them in Java programs. We aren't going to explore the details of defining objects for a while, but we want to begin using objects early. One of the most attractive features of Java is that it comes with a rich library of predefined objects that can be used to solve many common programming tasks.

The chapter concludes with an exploration of a very important kind of object known as a `Scanner`. Using a `Scanner` object, you can write programs that obtain values from the user. This feature will allow you to write interactive programs that prompt for input as well as producing output.

3.1 Parameters

Humans are very good at learning new tasks. When we learn, we often develop a single generalized solution for a family of related tasks. For example, someone might ask you to take 10 steps forward or 20 steps forward. These are different tasks, but they both involve taking a certain number of steps forward. We think of this action as a single task of taking steps forward, and we understand that the number of steps will vary from one task to another. In programming terms, we refer to the number of steps as a *parameter* that allows us to generalize the task.

> **Parameter (Parameterize)**
>
> Any of a set of characteristics that distinguish different members of a family of tasks. To parameterize a task is to identify a set of its parameters.

For a programming example, let's return to the `DrawFigure2` program of Chapter 2. It performs its task adequately, but there are several aspects of this program that we can improve. For example, there are many different places where a `for` loop writes out spaces. This approach is redundant and can be consolidated into a single method that performs all space-writing tasks.

Each space-writing task requires a different number of spaces, so you need some way to tell the method how many spaces to write. The methods you've written so far have a simple calling mechanism where you say:

```
writeSpaces();
```

One approach might be to set a variable to a particular value before the method is called:

```
int number = 10;
writeSpaces();
```

Then the method could look at the value of the variable `number` to see how many spaces to write. Unfortunately, this approach won't work. Recall from Chapter 2 that scope rules determine where variables can be accessed. Following those rules, the variable `number` would be a local variable in `main` that could not be seen inside `writeSpaces`.

Instead, you can specify one or more parameters to a method. The idea is that instead of writing a method that performs just one version of a task, you write a more flexible version that solves a family of related tasks that all differ by one or more parameters. In the case of the `writeSpaces` method, the parameter is the number of spaces to write.

The following is the definition of `writeSpaces` with a parameter for the number of spaces to write:

```
public static void writeSpaces(int number) {
    for (int i = 1; i <= number; i++) {
        System.out.print(" ");
    }
}
```

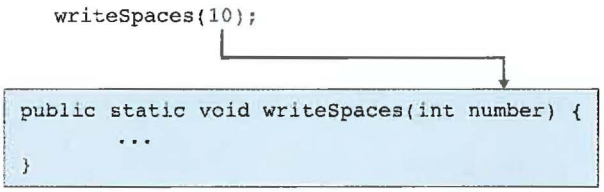
The parameter appears in the method header, after the name and inside the parentheses that you have, up to this point, been leaving empty. The `writeSpaces` method uses a parameter called `number` of type `int`. As we indicated earlier, you can no longer call the parameterized method by using just its name:

```
writeSpaces();
```

You must now say something like

```
writeSpaces(10);
```

When a call like this is made, the value `10` is used to initialize the `number` parameter. You can think of this as information flowing into the method from the call:



The parameter `number` is a local variable, but it gets its initial value from the call. Calling this method with the value `10` is equivalent to including the following declaration at the beginning of the `writeSpaces` method:

```
int number = 10;
```

Of course, this mechanism is more flexible than a specific variable declaration, because you can instead say

```
writeSpaces(20);
```

and it will be as if you had said

```
int number = 20;
```

at the beginning of the method. You can even use an integer expression for the call:

```
writeSpaces(3 * 4 - 5);
```

In this case, Java evaluates the expression to get the value `7` and then calls `writeSpaces`, initializing `number` to `7`.

Computer scientists use the word "parameter" broadly to mean both what appears in the method header (the *formal parameter*) and what appears in the method call (the *actual parameter*).

Formal Parameter

A variable that appears inside parentheses in the header of a method that is used to generalize the method's behavior.

Actual Parameter

A specific value or expression that appears inside parentheses in a method call.

The term "formal parameter" does not describe its purpose. A better name would be "generalized parameter." In the `writeSpaces` method, `number` is the generalized parameter that appears in the method declaration. It is a placeholder for some unspecified value. The values appearing in the method calls are the actual parameters, because each call indicates a specific task to perform. In other words, each call provides an actual value to fill the placeholder.

The word "argument" is often used as a synonym for "parameter," as in "These are the arguments I'm passing to this method." Some people prefer to reserve the word "argument" for actual parameters and the word "parameter" for formal parameters.

Let's look at an example of how you might use this `writeSpaces` method. Remember that the `DrawFigure2` program had the following method, called `drawTop`:

```
// produces the top half of the hourglass figure
public static void drawTop() {
    for (int line = 1; line <= SUB_HEIGHT; line++) {
        System.out.print("|");
        for (int i = 1; i <= (line - 1); i++) {
            System.out.print(" ");
        }
        System.out.print("\\");
        int dots = 2 * SUB_HEIGHT - 2 * line;
        for (int i = 1; i <= dots; i++) {
            System.out.print(".");
        }
        System.out.print("/");
        for (int i = 1; i <= (line - 1); i++) {
            System.out.print(" ");
        }
        System.out.println("|");
    }
}
```

Using the `writeSpaces` method, you can rewrite this as follows:

```
public static void drawTop() {
    for (int line = 1; line <= SUB_HEIGHT; line++) {
        System.out.print("|");
        writeSpaces(line - 1);
        System.out.print("\\");
        int dots = 2 * SUB_HEIGHT - 2 * line;
        for (int i = 1; i <= dots; i++) {
            System.out.print(".");
        }
        System.out.print("/");
        writeSpaces(line - 1);
        System.out.println("|");
    }
}
```

Notice that `writeSpaces` is called two different times, specifying how many spaces are required in each case. You could modify the `drawBottom` method from the `DrawFigure2` program similarly to simplify it.

The Mechanics of Parameters

When Java executes a call on a method, it initializes the method's parameters. For each parameter, it first evaluates the expression passed as the actual parameter and then uses the result to initialize the local variable whose name is given by the formal parameter. Let's use an example to clarify this process:

```
1  public class ParameterExample {
2      public static void main(String[] args) {
3          int spaces1 = 3;
4          int spaces2 = 5;
5
6          System.out.print("*");
7          writeSpaces(spaces1);
8          System.out.println("*");
9
10         System.out.print("!");
11         writeSpaces(spaces2);
12         System.out.println("!");
13
14         System.out.print("'");
15         writeSpaces(8);
16         System.out.println("'");
```

```
17
18          System.out.print("<");
19          writeSpaces(spaces1 * spaces2 - 5);
20          System.out.println(">");
21      }
22
23      // writes "number" spaces on the current output line
24      public static void writeSpaces(int number) {
25          for (int i = 1; i <= number; i++) {
26              System.out.print(" ");
27          }
28      }
29  }
```

In the first two lines of the `main` method, the computer finds instructions to allocate and initialize two variables:

The next three lines of code produce an output line with three spaces bounded by asterisks on either side:

```
System.out.print("*");
writeSpaces(spaces1);
System.out.println("*");
```

You can see where the asterisks come from, but look at the method call that produces the spaces. When Java executes the call on `writeSpaces`, it must set up its parameter. To set up the parameter, Java first evaluates the expression being passed as the actual parameter. The expression is simply the variable `spaces1`, which has the value `3`. Therefore, the expression evaluates to `3`. Java uses this result to initialize a local variable called `number`.

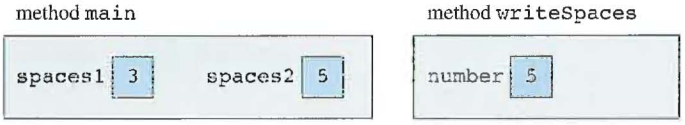
The following diagram indicates how the computer's memory would look as the `writeSpaces` method is entered the first time. Because there are two methods involved (`main` and `writeSpaces`), the diagram indicates which variables are local to `main` (`spaces1` and `spaces2`) and which are local to `writeSpaces` (the parameter `number`):

The net effect of this process is that the `writeSpaces` method has a local copy of the value stored in the variable `spaces1` from the `main` method. The `println` that comes after the call on `writeSpaces` puts an asterisk at the end of the line and then completes the line of output.

Let's now trace the next three lines of code:

```
System.out.print("!");
writeSpaces(spaces2);
System.out.println("!");
```

The first line prints an exclamation mark on the second line of output, then calls `writeSpaces` again, this time with the variable `spaces2` as its actual parameter. The computer evaluates this expression, obtaining the result `5`. This value is used to initialize `number`. Thus, this time it creates a copy of the value stored in the variable `spaces2` from the `main` method:

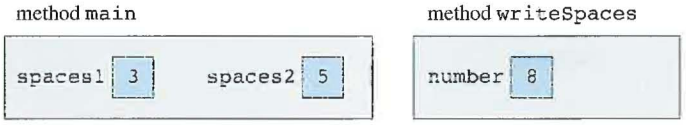


Because `number` has a different value this time (`5` instead of `3`), the method produces a different number of spaces. After the method executes, the `println` finishes the line of output with a second exclamation mark.

Here are the next three lines of code:

```
System.out.print("'");
writeSpaces(8);
System.out.println("'");
```

This code writes a single quotation mark at the beginning of the third line of output and then calls `writeSpaces` again. This time it uses the integer literal `8` as the expression, which means that it initializes the parameter `number` as a copy of the number 8:

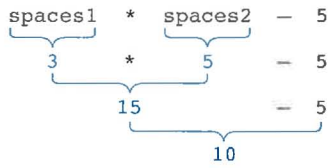


Again, the method will behave differently because of the different value of `number`. It prints eight spaces on the line and finishes executing. Then the `println` completes the line of output by printing another single quotation mark at the end of the line.

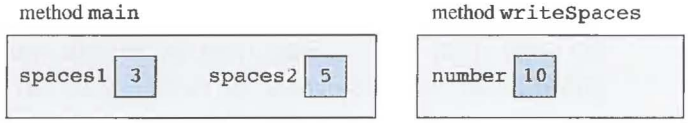
Finally, the last three lines of code in the `main` method are:

```
System.out.print("<");
writeSpaces(spaces1 * spaces2 - 5);
System.out.println(">");
```

This code prints a less-than character at the beginning of the fourth line of output and then makes a final call on the `writeSpaces` method. This time the actual parameter is an expression, not just a variable or literal value. Thus, before the call is made, the computer evaluates the expression to determine its value:



The computer uses this result to initialize `number`:



Now `number` is a copy of the value described by this complex expression. Therefore, the total output of this program is

```
*   *
!     !
'         '
<           >
```

Common Programming Error

Confusing Actual and Formal Parameters

Many students get used to seeing declarations of formal parameters and mistakenly believe that their syntax is identical to that for passing actual parameters. It's a common mistake to write the type of a variable as it's being passed to a parameter:

```
writeSpaces(int spaces1); // this doesn't work
```

Continued on next page

Continued from previous page

This confusion is due to the fact that parameters' types are written in the declaration of the method, like this:

```
public static void writeSpaces(int number)
```

Types must be written when variables or parameters are declared, but when variables are used, such as when the code calls a method and passes the variables as actual parameters, their types are not written. Actual parameters are not declarations; therefore, types should not be written before them:

```
writeSpaces(spaces1); // much better!
```

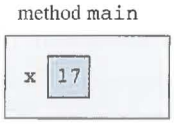
Limitations of Parameters

We've seen that a parameter can be used to provide input to a method. But while you can use a parameter to send a value into a method, you can't use a parameter to get a value out of a method.

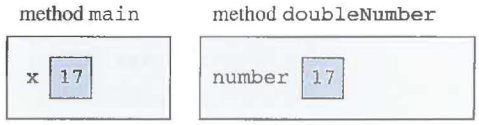
When a parameter is set up, a local variable is created and is initialized to the value being passed as the actual parameter. The net effect is that the local variable is a copy of the value coming from the outside. Since it is a local variable, it can't influence any variables outside the method. Consider the following sample program:

```
1  public class ParameterExample2 {
2      public static void main(String[] args) {
3          int x = 17;
4          doubleNumber(x);
5          System.out.println("x = " + x);
6          System.out.println();
7
8          int number = 42;
9          doubleNumber(number);
10         System.out.println("number = " + number);
11     }
12
13     public static void doubleNumber(int number) {
14         System.out.println("Initial value = " + number);
15         number = number * 2;
16         System.out.println("Final value = " + number);
17     }
18 }
```

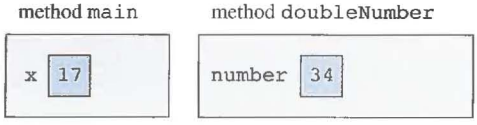
This program begins by declaring and initializing an integer variable called `x` with the value `17`:



It then calls the method `doubleNumber`, passing `x` as a parameter. The value of `x` is used to initialize the parameter `number` as a local variable of the method called `doubleNumber`:

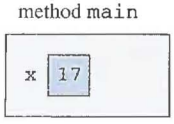


The program then executes the statements inside of `doubleNumber`. First, `doubleNumber` prints the initial value of `number` (`17`). Then it doubles `number`:

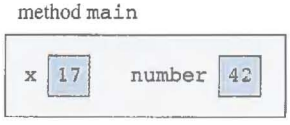


Notice that this has no effect on the variable `x`. The parameter called `number` is a copy of `x`, so even though they started out the same, changing the value of `number` does not affect `x`. Next, `doubleNumber` reports the new value of `number` (`34`).

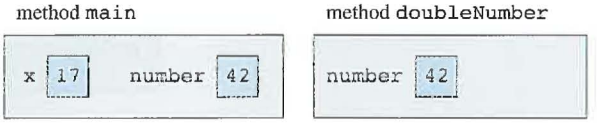
At this point, `doubleNumber` finishes executing and we return to `main`:



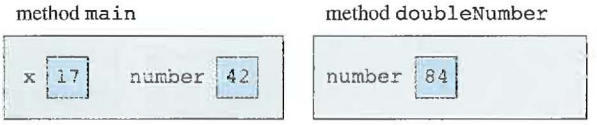
The next statement in the `main` method reports the value of `x`, which is `17`. Then it declares and initializes a variable called `number` with the value `42`:



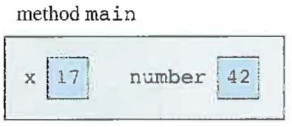
The following statement calls `doubleNumber` again, this time passing it the value of `number`. This is an odd situation because the parameter has the same name as the variable in `main`, but Java doesn't care. It always creates a new local variable for the `doubleNumber` method:



So, at this point there are two different variables called `number`, one in each method. Now it's time to execute the statements of `doubleNumber` again. It first reports the value of `number` (`42`), then doubles it:



Again, notice that doubling `number` inside `doubleNumber` has no effect on the original variable `number` in `main`. These are separate variables. The method then reports the new value of `number` (`84`) and returns to `main`:



The program then reports the value of `number` and terminates. So, the overall output of the program is as follows:

```
Initial value = 17
Final value = 34
x = 17

Initial value = 42
Final value = 84
number = 42
```

The local manipulations of the parameter do not change these variables outside the method. The fact that variables are copied is an important aspect of parameters. On the positive side, we know that the variables are protected from change because the parameters are copies of the originals. On the negative side, it means that although parameters will allow us to send values into a method, they will not allow us to get values back out of a method.

Multiple Parameters

So far, our discussion of parameter syntax has been informal. It's about time that we wrote down more precisely the syntax we use to declare static methods with parameters. Here it is:

```
public static void <name>(<type> <name>, ..., <type> <name>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

This template indicates that we can declare as many parameters as we want inside the parentheses that appear after the name of a method in its header. We use commas to separate different parameters.

As an example of a method with multiple parameters, let's consider a variation of `writeSpaces`. It is convenient that we can use the method to write different numbers of spaces, but it always writes spaces. What if we want 18 asterisks or 23 periods or 17 question marks? We can generalize the task even further by having the method take two parameters—a character and a number of times to write that character:

```
public static void writeChars(char ch, int number) {
    for (int i = 1; i <= number; i++) {
        System.out.print(ch);
    }
}
```

The character to be printed is a parameter of type `char`, which we will discuss in more detail in the next chapter. Recall that character literals are enclosed in single quotation marks.

The syntax template for calling a method that accepts parameters is the following:

```
<method name>(<expression>, <expression>, ..., <expression>);
```

By calling the `writeChars` method you can write code like the following:

```
writeChars('=', 20);
System.out.println();
for (int i = 1; i <= 10; i++) {
    writeChars('>', i);
    writeChars(' ', 20 - 2 * i);
    writeChars('<', i);
    System.out.println();
}
```

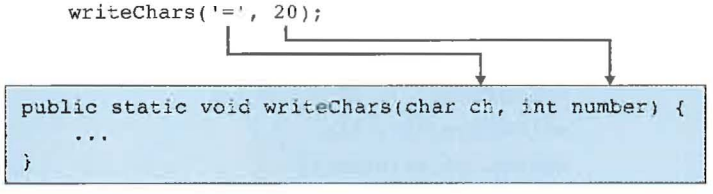
This code produces the following output:

```
====================
>                    <
>>                  <<
>>>                <<<
>>>>              <<<<
>>>>>            <<<<<
>>>>>>          <<<<<<
>>>>>>>        <<<<<<<
>>>>>>>>      <<<<<<<<
>>>>>>>>>    <<<<<<<<<
>>>>>>>>>><<<<<<<<<<
```

Using the `writeChars` method we can write an even better version of the `drawTop` method from the `DrawFigure2` program of Chapter 2. We saw earlier that using `writeChars` we could eliminate two of the inner loops, but this left us with the inner loop to print dots. Now we can eliminate all three of the inner loops and produce a much more readable version of the method:

```
public static void drawTop() {
    for (int line = 1; line <= SUB_HEIGHT; line++) {
        System.out.print("|");
        writeChars(' ', line - 1);
        System.out.print("\\");
        writeChars('.', 2 * SUB_HEIGHT - 2 * line);
        System.out.print("/");
        writeChars(' ', line - 1);
        System.out.println("|");
    }
}
```

You can include as many parameters as you want when you define a method. Each method call must provide exactly that number of parameters, in the same order. For example, consider the first call on `writeChars` in the preceding code fragment and the header for `writeChars`. Java lines up the parameters in sequential order (with the first actual parameter going into the first formal parameter and the second actual parameter going into the second formal parameter):



We've seen that methods can call other methods; this is equally true of methods that take parameters. For example, here is a method for drawing a box of a given height and width that calls the `writeChars` method:

```
public static void drawBox(int height, int width) {
    // draw top of box
    writeChars('*', width);
    System.out.println();

    // draw middle lines
    for (int i = 1; i <= height - 2; i++) {
        System.out.print('*');
        writeChars(' ', width - 2);
        System.out.println("*");
    }

    // draw bottom of box
    writeChars('*', width);
    System.out.println();
}
```

Notice that `drawBox` is passed values for its parameters called `height` and `width` and that these parameters are used to form expressions that are passed as values to `writeChars`. For example, inside the `for` loop we call `writeChars` asking it to produce `width - 2` spaces. (We subtract 2 because we print a star at the beginning and the end of the line.) Here is a sample call on the method:

```
drawBox(5, 10);
```

This code produces the following output:

```
**********
*        *
*        *
*        *
**********
```

When you're writing methods that accept many parameters, the method header can become very long. It is common to *wrap* long lines (ones that exceed roughly 80 characters in length) by inserting a line break after an operator or parameter and indenting the line that follows by twice the normal indentation width:

```
// this method's header is too long, so we'll wrap it
public static void printTriangle(int xCoord1, int yCoord1,
        int xCoord2, int yCoord2, int xCoord3, int yCoord3) {
    ...
}
```

Parameters versus Constants

In Chapter 2, you saw that class constants are a useful mechanism to increase the flexibility of your programs. By using such constants, you can make it easy to modify a program to behave differently. Parameters provide much of the same flexibility, and more. Consider the `writeSpaces` method. Suppose you wrote it using a class constant:

```
public static final int NUMBER_OF_SPACES = 10;
```

This approach would give you the flexibility to produce a different number of spaces, but it has one major limitation: The constant can change only from execution to execution; it cannot change within a single execution. In other words, you can execute the program once with one value, edit the program, recompile, and then execute it again with a different value, but you can't use different values in a single execution of the program using a class constant.

Parameters are more flexible. Because you specify the value to be used each time you call the method, you can use several different values in a single program execution. As you have seen, you can call the method many different times within a single program execution and have it behave differently every time. However, using parameters involves more work for the programmer than using class constants. It makes your method headers and method calls more tedious, not to mention making the execution (and, thus, the debugging) more complex.

Therefore, you will probably find occasion to use each technique. The basic rule is to use a class constant when you only want to change the value from execution to execution. If you want to use different values within a single execution, use a parameter.

Overloading of Methods

You'll often want to create slight variations of the same method, passing different parameters. For example, you could have a `drawBox` method that allows you to specify a particular height and width, but you might also want to have a version that draws a box of default size. In other words, sometimes you want to specify these values, as in

```
drawBox(8, 10);
```

and other times you want to just draw a box with the standard height and width:

```
drawBox();
```

Some programming languages require you to come up with different names for these versions, such as `drawBox` and `drawDefaultBox`. As you can imagine, coming up with new names for each variation becomes tedious. Fortunately, Java allows you to have more than one method with the same name, as long as they have different parameters. This is called *overloading*. The primary requirement for overloading is that the different methods that you define must have different *method signatures*.

Method Signature

The name of a method, along with its number and type of parameters.

Method Overloading

The ability to define two or more different methods with the same name but different method signatures.

The two example `drawBox` versions clearly have different method signatures, because one has two parameters and the other has zero parameters. It would be obvious from any call on the method which version to use: If you see two parameters, you execute the version with two parameters; if you see zero parameters, you execute the version with zero parameters.

The situation gets more complicated when overloading involves the same number of parameters, but this turns out to be one of the most useful applications of overloading. For example, the `println` method is actually a series of overloaded methods. We can call `println` passing it a `String`, an `int`, a `double`, and so on. This flexibility is implemented as a series of different methods, all of which take one parameter: One version takes a `String`, another version takes an `int`, another version takes a `double`, and so on. Obviously, you do slightly different things to print one of these kinds of data versus another, which is why it's useful to have these different versions of the method.

3.2 Methods That Return Values

The last few methods we've looked at have been action-oriented methods that perform some specific task. You can think of them as being like commands that you could give someone, as in "Draw a box" or "Draw a triangle." Parameters allow these commands to be more flexible, as in "Draw a box that is 10 by 20."

You will also want to be able to write methods that compute values. These methods are more like questions, as in "What is the square root of 2.5?" or "What do you get when you carry 2.3 to the 4th power?" Consider, for example, a method called `sqrt` that would compute the square root of a number.

It might seem that the way to write such a method would be to have it accept a parameter of type `double` and `println` its square root to the console. But you may want to use the square root as part of a larger expression or computation, such as solving a quadratic equation or computing the distance between points on an *x*/*y* plane.

A better solution would be a square root command that passes the number of interest as a parameter and returns its square root back to the program as a result. You could then use the result as part of an expression, store it in a variable, or print it to the console. Such a command is a new type of method that is said to *return* a value.

Return

To send a value out as the result of a method that can be used in an expression in your program. Void methods do not return any value.

If you had such a method, you could ask for the square root of 2.5 by writing code like this:

```
// assuming you had a method named sqrt
double answer = sqrt(2.5);
```

The `sqrt` method has a parameter (the number whose square root you want to find), and it also returns a value (the square root). The actual parameter `2.5` goes "into" the method, and the square root comes out. In the preceding code, the returned result is stored in a variable called `answer`.

You can tell whether or not a method returns a value by looking at its header. All the methods you've written so far have begun with `public static void`. The word `void` is known as the *return type* of the method.

The `void` return type is a little odd because, as its name implies, the method returns nothing. A method can return any legal type: an `int`, a `double`, or any other type. In the case of the `sqrt` method, you want it to return a `double`, so you would write its header as follows:

```
public static double sqrt(double n)
```

As in the previous case, the word that comes after `public static` is the return type of the method:

```
public static double sqrt(double n)
              ^^^^^^
           return type
```

Fortunately, you don't actually need to write a method for computing the square root of a number, because Java has one that is built in. The method is included in a class known as `Math` that includes many useful computing methods. So, before we discuss the details of writing methods that return values, let's explore the `Math` class and what it has to offer.

The `Math` Class

In Chapter 1 we mentioned that a great deal of predefined code, collectively known as the Java class libraries, has been written for Java. One of the most useful classes is `Math`. It includes predefined mathematical constants and a large number of common mathematical functions. The `Math` class should be available on any machine on which Java is properly installed.

As we noted in the previous section, the `Math` class has a method called `sqrt` that computes the square root of a number. The method has the following header:

```
public static double sqrt(double n)
```

This header says that the method is called `sqrt`, that it takes a parameter of type `double`, and that it returns a value of type `double`.

Unfortunately, you can't just call this method directly by referring to it as `sqrt` because it is in another class. Whenever you want to refer to something declared in another class, you use *dot notation:*

```
<class name>.<element>
```

So you would refer to this method as `Math.sqrt`. Here's a sample program that uses the method:

```
1  public class WriteRoots {
2      public static void main(String[] args) {
3          for (int i = 1; i <= 20; i++) {
4              double root = Math.sqrt(i);
5              System.out.println("sqrt(" + i + ") = " + root);
6          }
7      }
8  }
```

This program produces the following output:

```
sqrt(1) = 1.0
sqrt(2) = 1.4142135623730951
sqrt(3) = 1.7320508075688772
sqrt(4) = 2.0
sqrt(5) = 2.23606797749979
sqrt(6) = 2.449489742783178
sqrt(7) = 2.6457513110645907
sqrt(8) = 2.8284271247461903
sqrt(9) = 3.0
sqrt(10) = 3.1622776601683795
sqrt(11) = 3.3166247903554
sqrt(12) = 3.4641016151377544
sqrt(13) = 3.605551275463989
sqrt(14) = 3.7416573867739413
sqrt(15) = 3.872983346207417
sqrt(16) = 4.0
sqrt(17) = 4.123105625617661
sqrt(18) = 4.242640687119285
sqrt(19) = 4.358898943540674
sqrt(20) = 4.47213595499958
```

Table 3.1 Math Constants

Constant	Description
`E`	base used in natural logarithms (2.71828. . .)
`PI`	ratio of circumference of a circle to its diameter (3.14159 . . .)

Notice that we passed a value of type `int` to `Math.sqrt`, but the header says that it expects a value of type `double`. Remember that if Java is expecting a `double` and gets an `int`, it converts the `int` into a corresponding `double`.

The `Math` class also defines two frequently used constants: e and π (see Table 3.1). Following the Java convention, we use all uppercase letters for their names and refer to them as `Math.E` and `Math.PI`.

Table 3.2 lists some of the most useful static methods from the `Math` class. You can see a complete list of methods defined in the `Math` class by checking out the API

Table 3.2 Useful Static Methods in the `Math` Class

Method	Description	Example
`abs`	absolute value	`Math.abs(-308)` returns 308
`ceil`	ceiling (rounds upward)	`Math.ceil(2.13)` returns `3.0`
`cos`	cosine (radians)	`Math.cos(Math.PI)` returns `-1.0`
`exp`	exponent base e	`Math.exp(1)` returns `2.7182818284590455`
`floor`	floor (rounds downward)	`Math.floor(2.93)` returns `2.0`
`log`	logarithm base e	`Math.log(Math.E)` returns `1.0`
`log10`	logarithm base 10	`Math.log10(1000)` returns `3.0`
`max`	maximum of two values	`Math.max(45, 207)` returns 207
`min`	minimum of two values	`Math.min(3.8, 2.75)` returns `2.75`
`pow`	power (general exponentiation)	`Math.pow(3, 4)` returns `81.0`
`random`	random value	`Math.random()` returns a random `double` value k such that $0.0 \leq k < 1.0$
`round`	round real number to nearest integer	`Math.round(2.718)` returns 3
`sin`	sine (radians)	`Math.sin(0)` returns `0.0`
`sqrt`	square root	`Math.sqrt(2)` returns `1.4142135623730951`
`toDegrees`	converts from radians to degrees	`Math.toDegrees(Math.PI)` returns `180.0`
`toRadians`	converts from degrees to radians	`Math.toRadians(270.0)` returns `4.71238898038469`

documentation for your version of Java. The API describes how to use the standard libraries that are available to Java programmers. It can be a bit overwhelming, because the Java libraries are vast. Wander around a bit if you are so inclined, but don't be dismayed that there are so many libraries to choose from in Java.

If you do look into the `Math` API, you'll notice that the `Math` class has several overloaded methods. For example, there is a version of the absolute value method (`Math.abs`) for integers and another for `doubles`. The rules that govern which method is called are complex, so we won't cover them here. The basic idea, though, is that Java tries to find the method that is the best fit. For the most part, you don't have to think much about this issue; you can just let Java choose for you, and it will generally make the right choice.

Defining Methods That Return Values


VideoNote

You can write your own methods that return values by using a special statement known as a `return` statement. For example, we often use a method that returns a value to express an equation. There is a famous story about the mathematician Carl Friedrich Gauss that illustrates the use of such a method. When Gauss was a boy, his teacher asked the class to add up the integers 1 through 100, thinking that it would take a while for them to complete the task. Gauss immediately found a formula and presented his answer to the teacher. He used a simple trick of adding two copies of the series together, one in forward order and one in backward order. This method allowed him to pair up values from the two copies so that their sum was the same:

First series	Second series	Sum
1	100	101
2	99	101
3	98	101
4	97	101
. . .	. . .	. . .
100	1	101

Every entry in the right-hand column is equal to 101 and there are 100 rows in this table, so the overall sum is $100 \times 101 = 10{,}100$. Of course, that's the sum of two copies of the sequence, so the actual answer is half that. Using this approach, Gauss determined that the sum of the first 100 integers is 5050. When the series goes from 1 to n, the sum is $(n + 1) \times n / 2$.

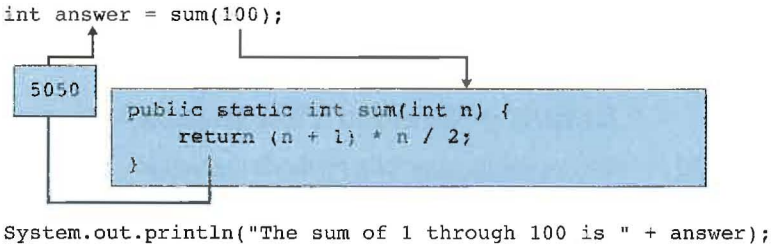
We can use Gauss' formula to write a method that computes the sum of the first n integers:

```
public static int sum(int n) {
    return (n + 1) * n / 2;
}
```

The sum method could be used by the main method in code such as the following:

```
int answer = sum(100);
System.out.println("The sum of 1 through 100 is " + answer);
```

A diagram of what happens when this code is executed follows. The method is invoked with the parameter n being initialized to 100. Plugging this value into the formula, we get a value of 5050, which is sent back to be stored in the variable called answer:



Notice once again that in the header for the method the familiar word void (indicating no return value) has been replaced with the word int. Remember that when you declare a method that returns a value, you have to tell Java what kind of value it will return. Thus, we can update our syntax template for static methods once more to clarify that the header includes a return type (void for none):

```
public static <type> <name>(<type> <name>, ..., <type> <name>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The syntax of the return statement is:

```
return <expression>;
```

When Java encounters a return statement, it evaluates the given expression and immediately terminates the method, returning the value it obtained from the expression. As a result, it's not legal to have any other statements after a return statement; the return must be the last statement in your method. It is also an error for a Java method with a return type other than void to terminate without a return.

There are exceptions to the previous rules, as you'll see later. For example, it is possible for a method to have more than one return statement; this will come up in the next chapter, when we discuss conditional execution using if and if/else statements.

Let's look at another example method that returns a value. In *The Wizard of Oz*, the Scarecrow after being given a diploma demonstrates his intelligence by saying, "The sum of the square roots of any two sides of an isosceles triangle is equal to the square root of the remaining side. Oh, joy, oh, rapture. I've got a brain!" Probably he

was trying to state the Pythagorean theorem, although it's not clear whether the writers were bad at math or whether they were making a comment about the value of a diploma. In an episode of *The Simpsons*, Homer repeats the Scarecrow's mistaken formula after putting on a pair of Henry Kissinger's glasses that he finds in a bathroom at the Springfield nuclear power plant.

The correct Pythagorean theorem refers only to right triangles and says that the length of the hypotenuse of a right triangle is equal to the square root of the sums of the squares of the two remaining sides. If you know the lengths of two sides a and b of a right triangle and want to find the length of the third side c, you compute it as follows:

$$c = \sqrt{a^2 + b^2}$$

Common Programming Error

Ignoring the Returned Value

When you call a method that returns a value, the expectation is that you'll do something with the value that's returned. You can print it, store it in a variable, or use it as part of a larger expression. It is legal (but unwise) to simply call the method and ignore the value being returned from it:

```
sum(1000);   // doesn't do anything
```

The preceding call doesn't print the sum or have any noticeable effect. If you want the value printed, you must include a `println` statement:

```
int answer = sum(1000);     // better
System.out.println("Sum up to 1000 is " + answer);
```

A shorter form of the fixed code would be the following:

```
System.out.println("Sum up to 1000 is " + sum(1000));
```

Say you want to print out the lengths of the hypotenuses of two right triangles, one with side lengths of 5 and 12, and the other with side lengths of 3 and 4. You could write code such as the following:

```
double c1 = Math.sqrt(Math.pow(5, 2) + Math.pow(12, 2));
System.out.println("hypotenuse 1 = " + c1);
double c2 = Math.sqrt(Math.pow(3, 2) + Math.pow(4, 2));
System.out.println("hypotenuse 2 = " + c2);
```

The preceding code is correct, but it's a bit hard to read, and you'd have to duplicate the same complex math a third time if you wanted to include a third triangle. A better solution would be to create a method that computes and returns the hypotenuse length when given the lengths of the two other sides as parameters. Such a method would look like this:

```
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
    return c;
}
```

This method can be used to craft a more concise and readable `main` method, as shown here.

```
1  public class Triangles {
2      public static void main(String[] args) {
3          System.out.println("hypotenuse 1 = " + hypotenuse(5, 12));
4          System.out.println("hypotenuse 2 = " + hypotenuse(3, 4));
5      }
6
7      public static double hypotenuse(double a, double b) {
8          double c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
9          return c;
10     }
11 }
```

A few variations of this program are possible. For one, it isn't necessary to store the `hypotenuse` method's return value into the variable `c`. If you prefer, you can simply compute and return the value in one line. In this case, the body of the `hypotenuse` method would become the following:

```
return Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
```

Also, some programmers avoid using `Math.pow` for low powers such as 2 and just manually do the multiplication. Using that approach, the body of the hypotenuse method would look like this:

```
return Math.sqrt(a * a + b * b);
```

Common Programming Error

Statement after Return

It's illegal to place other statements immediately after a `return` statement, because those statements can never be reached or executed. New programmers often accidentally do this when trying to print the value of a variable after returning. Say you've written the `hypotenuse` method but have accidentally written the parameters to `Math.pow` in the wrong order, so the method is not producing the right answer. You would try to debug this by printing the value of `c` that is being returned. Here's the faulty code:

```
// trying to find the bug in this buggy version of hypotenuse
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(2, a) + Math.pow(2, b));
    return c;
    System.out.println(c);  // this doesn't work
}
```

The compiler complains about the `println` statement being unreachable, since it follows a `return` statement. The compiler error output looks something like this:

```
Triangles.java:10: error: unreachable statement
        System.out.println(c);
        ^
Triangles.java:11: error: missing return statement
    }
    ^
2 errors
```

The fix is to move the `println` statement earlier in the method, before the `return` statement:

```
public static double hypotenuse(double a, double b) {
    double c = Math.sqrt(Math.pow(2, a) + Math.pow(2, b));
    System.out.println(c);   // better
    return c;
}
```

3.3 Using Objects

We've spent a considerable amount of time discussing the primitive types in Java and how they work, so it's about time that we started talking about objects and how they work.

The idea for objects came from the observation that as we start working with new kinds of data (integers, reals, characters, text, etc.), we find ourselves writing a lot of

methods that operate on that data. Rather than completely separating the basic operations from the data, it seemed to make sense to include them together. This packaging of data and operations into one entity is the central idea behind objects. An *object* stores some data and has methods that act on its data.

Object

A programming entity that contains state (data) and behavior (methods).

As we said in Chapter 1, classes are the basic building blocks of Java programs. But classes also serve another purpose: to describe new types of objects.

Class

A category or type of object.

When it is used this way, a class is like a blueprint of what the object looks like. Once you've given Java the blueprint, you can ask it to create actual objects that match that blueprint. We sometimes refer to the individual objects as *instances* of the class. We tend to use the words "instance" and "object" interchangeably.

This concept is difficult to understand in the abstract. To help you come to grips with what it means and how it works, we'll look at several different classes. In keeping with our idea of focusing on fundamental concepts first, in this chapter we'll study how to use existing objects that are already part of Java, but we aren't going to study how to define our own new types of objects just yet. We'll get to that in Chapter 8, after we've had time to practice using objects.

Using objects differs from using primitive types, so we'll have to introduce some new syntax and concepts. It would be nice if Java had a consistent model for using all types of data, but it doesn't. Consequently, if you want to understand how your programs operate, you'll have to learn two sets of rules: one for primitives and one for objects.

`String` Objects

`String` objects are one of the most useful and most commonly used types of objects in Java, so they make a good starting point. They aren't the best example of objects, though, because there are a lot of special rules that apply only to `strings`. In the next section, we'll look at a more typical kind of object.

One special property of `String` objects is that there are literals that represent them (string literals). We've been using them in `println` statements since Chapter 1. What we haven't discussed is that these literal values represent objects of type `String` (instances of the `String` class). For example, in the same way that you can say

```
int x = 8;
```

you can say

```
String s = "hello there";
```

You can declare variables of type `String` and use the assignment statement to give values to these variables. You can also write code that involves `String` expressions:

```
String s1 = "hello";
String s2 = "there";
String combined = s1 + " " + s2;
```

This code defines two `String`s that each represent a single word and a third `String` that represents the concatenation of the two words with a space in between. You'll notice that the type `String` is capitalized (as are the names of all object types in Java), unlike the primitive types such as `double` and `int`.

These examples haven't shown what's special about `String` objects, but we're getting there. Remember that the idea behind objects was to include basic operations with the data itself, the way we make cars that have controls built in. The data stored in a `String` is a sequence of characters. There are all sorts of operations you might want to perform on this sequence of characters. For example, you might want to know how many characters there are in the `String`. `String` objects have a `length` method that returns this information.

If the `length` method were static, you would call it by saying something like

```
length(s)     // this isn't legal
```

But when you perform operations on objects, you use a different syntax. Objects store data and methods, so the method to report a `String`'s length actually exists inside that `String` object itself. To call an object's method, you write the name of the variable first, followed by a dot, and then the name of the method:

```
s.length()
```

Think of it as talking to the `String` object. When you ask for `s.length()`, you're saying, "Hey, `s`. I'm talking to you. What's your length?" Of course, different `String` objects have different lengths, so you will get different answers when you communicate with different `String` objects.

The general syntax for calling a method of an object is the following:

```
<variable>.<method name>(<expression>, <expression>, ..., <expression>)
```

For example, suppose that you have initialized two `String` variables as follows:

```
String s1 = "hello";
String s2 = "how are you?";
```

You can use a `println` to examine the length of each `String`:

```
System.out.println("Length of s1 = " + s1.length());
System.out.println("Length of s2 = " + s2.length());
```

This code produces the following output:

```
Length of s1 = 5
Length of s2 = 12
```

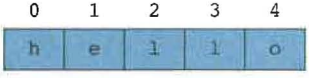
What else might you want to do with a `String` object? With the `length` method, you can figure out how many characters there are in a `String`, but what about getting the individual characters themselves? There are several ways to do this, but one of the most common is to use a method called `charAt` that returns the character at a specific location in the string.

This leads us to the problem of how to specify locations in a sequence. Obviously there is a first character, a second character, and so on, so it makes sense to use an integer to refer to a specific location. We call this the *index*.

Index

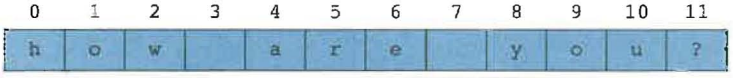
An integer used to specify a location in a sequence of values. Java generally uses zero-based indexing (with 0 as the first index value, followed by 1, 2, 3, and so on).

Each character of a `String` object is assigned an index value, starting with 0. For example, for the variable `s1` that refers to the string `"hello"`, the indexes are:



It may seem intuitive to consider the letter "h" to be at position 1, but there are advantages to starting with an index of 0. It's a convention that was adopted by the designers of the C language and that has been followed by the designers of C++ and Java, so it's a convention you'll have to learn to live with.

For the longer `String s2`, the positions are:



Notice that the spaces in this `String` have positions as well (here, positions 3 and 7). Also notice that the indexes for a given string always range from 0 to one less than the length of the string.

Using the `charAt` method, you can request specific characters of a string. The return type is `char`. For example, if you ask for `s1.charAt(1)` you'll get `'e'` (the 'e' in "hello"). If you ask for `s2.charAt(5)`, you'll get `'r'` (the 'r' in "how are you?"). For any `String`, if you ask for `charAt(0)`, you'll get the first character of the string.

When you are working with `String` objects, you'll often find it useful to write a `for` loop to handle the different characters of the `String`. Because `Strings` are

indexed starting at 0, this task is easier to write with `for` loops that start with `0` rather than `1`. Consider, for example, the following code that prints out the individual characters of `s1`:

```
String s1 = "hello";
for (int i = 0; i < s1.length(); i++) {
    System.out.println(i + ": " + s1.charAt(i));
}
```

This code produces the following output:

```
0: h
1: e
2: l
3: l
4: o
```

Remember that when we start loops at 0, we usually test with less than (<) rather than less than or equal to (<=). The string `s1` has five characters in it, so the call on `s1.length()` will return `5`. But because the first index is 0, the last index will be one less than 5, or 4. This convention takes a while to get used to, but zero-based indexing is used throughout Java, so you'll eventually get the hang of it.

Another useful `String` method is the `substring` method. It takes two integer arguments representing a starting and ending index. When you call the `substring` method, you provide two of these indexes: the index of the first character you want and the index just past the last index that you want.

Recall that the `String s2` has the following positions:

If you want to pull out the individual word "how" from this string, you'd ask for

```
s2.substring(0, 3)
```

Remember that the second value that you pass to the `substring` method is supposed to be one beyond the end of the substring you are forming. So, even though there is a space at position 3 in the original string, it will not be part of what you get from the call on `substring`. Instead, you'll get all the characters just before position 3.

Following this rule means that sometimes you will give a position to a substring at which there is no character. For instance, the last character in the string to which `s2` refers is at index 11 (the question mark). If you want to get the substring "you?" including the question mark, you'd ask for

```
s2.substring(8, 12)
```

There is no character at position 12 in `s2`, but this call asks for characters starting at position 8 that come before position 12, so this actually makes sense.

You have to be careful about what indexes you use, though. With the `substring` method you can ask for the position just beyond the end of the `String`, but you can't ask for anything beyond that. For example, if you ask for

```
s2.substring(8, 13)     // out of bounds!
```

your program will generate an execution error. Similarly, if you ask for the `charAt` at a nonexistent position, your program will generate an execution error. These errors are known as *exceptions.*

Exceptions are runtime errors as mentioned in Chapter 1.

Exception

A runtime error that prevents a program from continuing its normal execution.

We say that an exception is *thrown* when an error is encountered. When an exception is thrown, Java looks to see if you have written code to handle it. If not, program execution is halted and you will see what is known as a *stack trace* or *back trace.* The stack trace shows you the series of methods that have been called, in reverse order. In the case of bad `String` indexes, the exception prints a message such as the following to the console:

```
Exception in thread "main"
        java.lang.StringIndexOutOfBoundsException:
        String index out of range: 13
        at java.lang.String.substring(Unknown Source)
        at ExampleProgram.main(ExampleProgram.java:3)
```

You can use `Strings` as parameters to methods. For example, the following program uses `String` parameters to eliminate some of the redundancy in a popular children's song:

```
1  public class BusSong {
2      public static void main(String[] args) {
3          verse("wheels", "go", "round and round");
4          verse("wipers", "go", "swish, swish, swish");
5          verse("horn", "goes", "beep, beep, beep");
6      }
7
8      public static void verse(String item, String verb, String sound) {
9          System.out.println("The " + item + " on the bus " +
10                             verb + " " + sound + ",");
```

```
11          System.out.println(sound + ",");
12          System.out.println(sound + ".");
13          System.out.println("The " + item + " on the bus " +
14                             verb + " " + sound + ",");
15          System.out.println("All through the town.");
16          System.out.println();
17      }
18  }
```

It produces the following output:

```
The wheels on the bus go round and round,
round and round,
round and round.
The wheels on the bus go round and round,
All through the town.

The wipers on the bus go swish, swish, swish,
swish, swish, swish,
swish, swish, swish.
The wipers on the bus go swish, swish, swish,
All through the town.

The horn on the bus goes beep, beep, beep,
beep, beep, beep,
beep, beep, beep.
The horn on the bus goes beep, beep, beep,
All through the town.
```

Table 3.3 lists some of the most useful methods that you can call on `String` objects.

`Strings` in Java are *immutable,* which means that once they are constructed, their values can never be changed.

Immutable Object

An object whose value cannot be changed.

It may seem odd that `Strings` are immutable and yet have methods like `toUpperCase` and `toLowerCase`. But if you read the descriptions in the table carefully, you'll see that these methods don't actually change a given `String` object; instead they return a new string. Consider the following code:

```
String s = "Hello, Maria";
s.toUpperCase();
System.out.println(s);
```

Table 3.3 Useful Methods of `String` Objects

Method	Description	Example (assuming `s` is `"hello"`)
`charAt(index)`	character at a specific index	`s.charAt(1)` returns `'e'`
`endsWith(text)`	whether or not the string ends with some text	`s.endsWith("llo")` returns `true`
`indexOf(text)`	index of a particular character or `String` (−1 if not present)	`s.indexOf("o")` returns 4
`length()`	number of characters in the string	`s.length()` returns 5
`startsWith(text)`	whether or not the string starts with some text	`s.startsWith("hi")` returns `false`
`substring(start, stop)`	characters from start index to just before stop index	`s.substring(1, 3)` returns `"el"`
`toLowerCase()`	a new string with all lowercase letters	`s.toLowerCase()` returns `"hello"`
`toUpperCase()`	a new string with all uppercase letters	`s.toUpperCase()` returns `"HELLO"`

You might think that this will turn the string `s` into its uppercase equivalent, but it doesn't. The second line of code constructs a new string that has the uppercase equivalent of the value of `s`, but we don't do anything with this new value. In order to turn the string into uppercase, the key is to either store this new string in a different variable or reassign the variable `s` to point to the new string:

```
String s = "Hello, Maria";
s = s.toUpperCase();
System.out.println(s);
```

This version of the code produces the following output:

```
HELLO, MARIA
```

The `toUpperCase` and `toLowerCase` methods are particularly helpful when you want to perform string comparisons in which you ignore the case of the letters involved.

Interactive Programs and Scanner Objects

As you've seen, you can easily produce output in the console window by calling `System.out.println` and `System.out.print`. You can also write programs that pause and wait for the user to type a response. Such programs are known as *interactive* programs, and the responses typed by the user are known as *console input*.

Console Input

Responses typed by the user when an interactive program pauses for input.

When you refer to `System.out`, you are accessing an object in the `System` class known as the standard output stream, or "standard out" for short. There is a corresponding object for standard input known as `System.in`, but Java wasn't designed for console input, and `System.in` has never been particularly easy to use for this purpose. Fortunately for us, there is an easier way to read console input: `Scanner` objects.

Most objects have to be explicitly constructed by calling a special method known as a *constructor.*

Constructor (Construct)

A method that creates and initializes an object. Objects in Java programs must be constructed before they can be used.

Remember that a class is like a blueprint for a family of objects. Calling a constructor is like sending an order to the factory asking it to follow the blueprint to get you an actual object that you can manipulate. When you send in your order to the factory, you sometimes specify certain parameters (e.g., what color you want the object to be).

In Java, constructors are called using the special keyword `new`, followed by the object's type and any necessary parameters. For example, to construct a specific `Scanner` object, you have to pass information about the source of input. In particular, you have to provide an input stream. To read from the console window, pass it `System.in`:

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

Once you've constructed the `Scanner`, you can ask it to return a value of a particular type. A number of methods, all beginning with the word "next," are available to obtain the various types of values. Table 3.4 lists them.

Typically, you will use variables to keep track of the values returned by these methods. For example, you might say:

```
int n = console.nextInt();
```

The call on the `console` object's `nextInt` method pauses for user input. Whenever the computer pauses for input, it will pause for an entire line of input. In other words, it will wait until the user hits the Enter key before continuing to execute the program.

Table 3.4 Scanner Methods

Method	Description
`next()`	reads and returns the next token as a `String`
`nextDouble()`	reads and returns a `double` value
`nextInt()`	reads and returns an `int` value
`nextLine()`	reads and returns the next line of input as a `String`

You can use the `Scanner` class to read input line by line using the `nextLine` method, although we won't be using `nextLine` very much for now. The other "next" methods are all *token*-based (that is, they read single elements of input rather than entire lines).

Token

A single element of input (e.g., one word, one number).

By default, the `Scanner` uses *whitespace* to separate tokens.

Whitespace

Spaces, tab characters, and new line characters.

A `Scanner` object looks at what the user types and uses the whitespace on the input line to break it up into individual tokens. For example, the line of input

```
hello     there. "how are"        "you?"  all-one-token
```

would be split into six tokens:

```
hello
there.
"how
are"
"you?"
all-one-token
```

Notice that the `Scanner` includes punctuation characters such as periods, question marks, and quotation marks in the tokens it generates. It also includes dashes, so because there is no whitespace in the middle to break it up into different tokens, we get just one token for "all-one-token." It's possible to control how a `Scanner` turns things into tokens (a process called *tokenizing* the input), but we won't be doing anything that fancy.

It is possible to read more than one value from a `Scanner`, as in:

```
double x = console.nextDouble();
double y = console.nextDouble();
```

Because there are two different calls on the `console` object's `nextDouble` method, this code will cause the computer to pause until the user has entered two numeric values. The values can be entered on the same line or on separate lines. In general, the computer continues to pause for user input until it has obtained whatever values you have asked the `Scanner` to obtain.

If a user types something that isn't an integer when you call `nextInt`, such as `XYZZY`, the `Scanner` object generates an exception. Recall from the section on `String` objects that exceptions are runtime errors that halt program execution. In this case, you'll see runtime error output such as the following:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at Example.main(Example.java:13)
```

You will see in a later chapter how to test for user errors. In the meantime, we will assume that the user provides appropriate input.

Sample Interactive Program

Using the `Scanner` class, we can write a complete interactive program that performs a useful computation for the user. If you ever find yourself buying a house, you'll want to know what your monthly mortgage payment is going to be. The following is a complete program that asks for information about a loan and prints the monthly payment:

```
1   // This program prompts for information about a loan and
2   // computes the monthly mortgage payment.
3
4   import java.util.*;   // for Scanner
5
6   public class Mortgage {
7       public static void main(String[] args) {
8           Scanner console = new Scanner(System.in);
9
10          // obtain values
11          System.out.println("This program computes monthly " +
12                             "mortgage payments.");
13          System.out.print("loan amount     : ");
14          double loan = console.nextDouble();
15          System.out.print("number of years : ");
16          int years = console.nextInt();
17          System.out.print("interest rate   : ");
18          double rate = console.nextDouble();
19          System.out.println();
20
21          // compute result and report
22          int n = 12 * years;
23          double c = rate / 12.0 / 100.0;
```

```
24          double payment = loan * c * Math.pow(1 + c, n) /
25                                      (Math.pow(1 + c, n) - 1);
26          System.out.println("payment = $" + (int) payment);
27      }
28  }
```

The following is a sample execution of the program (user input is in bold):

```
This program computes monthly mortgage payments.
loan amount     : 275000
number of years : 30
interest rate   : 4.25

payment = $1352
```

The first thing we do in the program is construct a `Scanner` object, which we will use for console input. Next, we explain what the program is going to do, printing a description to the console. This is essential for interactive programs. You don't want a program to pause for user input until you've explained to the user what is going to happen.

Below the `println`, you'll notice several pairs of statements like these:

```
System.out.print("loan amount     : ");
double loan = console.nextDouble();
```

The first statement is called a *prompt,* a request for information from the user. We use a `print` statement instead of a `println` so that the user will type the values on the same line as the prompt (i.e., to the right of the prompt). The second statement calls the `nextDouble` method of the `console` object to read a value of type `double` from the user. This value is stored in a variable called `loan`. This pattern of prompt/read statements is common in interactive programs.

After prompting for values, the program computes several values. The formula for computing monthly mortgage payments involves the loan amount, the total number of months involved (a value we call `n`), and the monthly interest rate (a value we call `c`). The payment formula is given by the following equation:

$$payment = loan \frac{c(1 + c)^n}{(1 + c)^n - 1}$$

You will notice in the program that we use the `Math.pow` method for exponentiation to translate this formula into a Java expression.

The final line of the program prints out the monthly payment. You might imagine that we would simply say:

```
System.out.println("payment = $" + payment);
```

However, because the payment is stored in a variable of type `double`, such a statement would print all the digits of the number. For example, for the log listed above, it would print the following:

```
payment = $1352.8347004685593
```

That is a rather strange-looking output for someone who is used to dollars and cents. For the purposes of this simple program, it's easy to cast the `double` to an `int` and report just the dollar amount of the payment:

```
System.out.println("payment = $" + (int) payment);
```

Most people trying to figure out their mortgage payments aren't that interested in the pennies, so the program is still useful. In the next section, we will see how to round a `double` to two decimal places.

There is something new at the beginning of this class file called an *import declaration.* Remember that Java has a large number of classes included in what are collectively known as the Java class libraries. To help manage these classes, Java provides an organizational unit known as a *package.* Related classes are combined together into a single package. For example, the `Scanner` class is stored in a package known as `java.util`. Java programs don't normally have access to a package unless they include an import declaration.

Package

A collection of related Java classes.

Import Declaration

A request to access a specific Java package.

We haven't needed an `import` declaration yet because Java automatically imports every class stored in a package called `java.lang`. The `java.lang` package includes basic classes that most Java programs are likely to use (e.g., `System`, `String`, `Math`). Because Java does not automatically import `java.util`, you have to do it yourself.

Java allows you to use an asterisk to import all classes from a package:

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

But some people prefer to specifically mention each class they import. The import declaration allows you to import just a single class from a package, as in

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

The problem is that once you start importing one class from a package, you're likely to want to import others as well. We will use the asterisk version of `import` in this book to keep things simple.

3.4 Case Study: Projectile Trajectory

It's time to pull together the threads of this chapter with a more complex example that will involve parameters, methods that return values, mathematical computations, and the use of a `Scanner` object for console input.

Physics students are often asked to calculate the trajectory that a projectile will follow, given its initial velocity and its initial angle relative to the horizontal. For example, the projectile might be a football that someone has kicked. We want to compute the path it follows given Earth's gravity. To keep the computation reasonable, we will ignore air resistance.

There are several questions relating to this problem that we might want to answer:

- When does the projectile reach its highest point?
- How high does it reach?
- How long does it take to come back to the ground?
- How far does it land from where it was launched?

There are several ways to answer these questions. One simple approach is to provide a table that displays the trajectory step by step, indicating the x position, y position, and elapsed time.

To make such a table, we need to obtain three values from the user: the initial velocity, the angle relative to the horizontal, and the number of steps to include in the table we will produce. We could ask for the velocity in either meters/second or feet/second, but given that this is a physics problem, we'll stick to the metric system and ask for meters/second.

We also have to think about how to specify the angle. Unfortunately, most of the Java methods that operate on angles require angles in radians rather than degrees. We could request the angle in radians, but that would be highly inconvenient for the user, who would be required to make the conversion. Instead, we can allow the user to enter the angle in degrees and then convert it to radians using the built-in method `Math.toRadians`.

So, the interactive part of the program will look like this:

```
Scanner console = new Scanner(System.in);
System.out.print("velocity (meters/second)? ");
double velocity = console.nextDouble();
System.out.print("angle (degrees)? ");
double angle = Math.toRadians(console.nextDouble());
System.out.print("number of steps to display? ");
int steps = console.nextInt();
```

Notice that for the velocity and angle we call the `nextDouble` method of the `console` object, because we want to let the user specify any number (including one with a decimal point), but for the number of steps we call `nextInt`, because the number of lines in our table needs to be an integer.

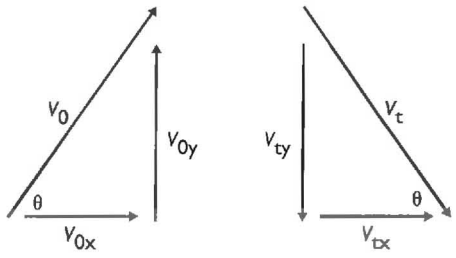


Figure 3.1 Initial and Final Velocity of Projectile

Look more closely at this line of code:

```
double angle = Math.toRadians(console.nextDouble());
```

Some beginners would write this as two separate steps:

```
double angleInDegrees = console.nextDouble();
double angle = Math.toRadians(angleInDegrees);
```

Both approaches work and are reasonable, but keep in mind that you don't need to divide this operation into two separate steps. You can write it in the more compact form as a single line of code.

Once we have obtained these values from the user, we are ready to begin the computations for the trajectory table. The *x*/*y*-position of the projectile at each time increment is determined by its velocity in each dimension and by the acceleration on the projectile due to gravity. Figure 3.1 shows the projectile's initial velocity v_0 and angle Θ just as it is thrown and final velocity v_t just as it hits the ground.

We need to compute the ***x*** component of the velocity versus the *y* component of the velocity. From physics, we know that these can be computed as follows:

```
double xVelocity = velocity * Math.cos(angle);
double yVelocity = velocity * Math.sin(angle);
```

Because we are ignoring the possibility of air resistance, the *x*-velocity will not change. The *y*-velocity, however, is subject to the pull of gravity. Physics tells us that on the surface of the Earth, acceleration due to gravity is approximately 9.81 meters/second^2. This is an appropriate value to define as a class constant:

```
public static final double ACCELERATION = -9.81;
```

Notice that we define gravity acceleration as a negative number because it decreases the *y*-velocity of an object (pulling it down as opposed to pushing it away).

Our goal is to display *x*, *y*, and elapsed time as the object goes up and comes back down again. The *y*-velocity decreases steadily until it becomes 0. From physics, we

know that the graph of the projectile will be symmetrical. The projectile will go upward until its y-velocity reaches 0, and then it will follow a similar path back down that takes an equal amount of time. Thus, the total time involved in seconds can be computed as follows:

```
double totalTime = -2.0 * yVelocity / ACCELERATION;
```

Now, how do we compute the values of x, y, and elapsed time to include in our table? It is relatively simple to compute two of these. We want steady time increments for each entry in the table, so we can compute the time increment by dividing the total time by the number of steps we want to include in our table:

```
double timeIncrement = totalTime / steps;
```

As noted earlier, the x-velocity does not change, so for each of these time increments, we move the same distance in the x-direction:

```
double xIncrement = xVelocity * timeIncrement;
```

The tricky value to compute here is the y-position. Because of acceleration due to gravity, the y-velocity changes over time. But from physics, we have the following general formula for computing the displacement of an object given the velocity v, time t, and acceleration a:

$$\text{displacement} = vt + \frac{1}{2}at^2$$

In our case, the velocity we want is the y-velocity and the acceleration is from the Earth's gravity constant. Here, then, is a pseudocode description of how to create the table:

```
set all of x, y, and t to 0.
for (given number of steps) {
    add timeIncrement to t.
    add xIncrement to x.
    reset y to yVelocity * t + 0.5 * ACCELERATION * t * t.
    report step #, x, y, t.
}
```

We are fairly close to having real Java code here, but we have to think about how to report the values of `x`, `y`, and `t` in a table. They will all be of type `double`, which means they are likely to produce a large number of digits after the decimal point. But we aren't interested in seeing all those digits, because they aren't particularly relevant and because our computations aren't that accurate.

Before we try to complete the code for the table, let's think about the problem of displaying only some of the digits of a number. The idea is to truncate the digits so

that we don't see all of them. One way to truncate is to cast a `double` to an `int`, which truncates all of the digits after the decimal point. We could do that, but we probably want at least some of those digits. Say, for example, that we want to report two digits after the decimal point. The trick is to bring the two digits we want to the other side of the decimal point. We can do that by multiplying by 100 and then casting to `int`:

```
(int) (n * 100.0)
```

This expression gets us the digits we want, but now the decimal point is in the wrong place. For example, if `n` is initially `3.488834`, the preceding expression will give us `348`. We have to divide this result by 100 to turn it back into the number 3.48:

```
(int) (n * 100.0) / 100.0
```

While we're at it, we can make one final improvement. Notice that the original number was `3.488834`. If we do simple truncation we get `3.48`, but really this number is closer to `3.49`. We can round to the nearest digit by calling the `Math.round` method on the number instead of casting it:

```
Math.round(n * 100.0) / 100.0
```

This is an operation that we are likely to want to perform on more than one number, so it deserves to be included in a method:

```
public static double round2(double n) {
    return Math.round(n * 100.0) / 100.0;
}
```

Getting back to our pseudocode for the table, we can incorporate calls on the `round2` method to get a bit closer to actual Java code:

```
set all of x, y, and t to 0.
for (given number of steps) {
    add timeIncrement to t.
    add xIncrement to x.
    reset y to yVelocity * t + 0.5 * ACCELERATION * t * t.
    report step #, round2(x), round2(y), round2(t).
}
```

It would be nice if the values in the table were aligned. To get numbers that line up nicely, we would have to use formatted output, which we will discuss in Chapter 4. For now, we can at least get the numbers to line up in columns by separating them with tab characters. Remember that the escape sequence `\t` represents a single tab.

If we're going to have a table with columns, it also makes sense to have a header for the table. And we probably want to include a line in the table showing the initial

condition, where x, y, and time are all equal to 0. So we can expand our pseudocode into the following Java code:

```
double x = 0.0;
double y = 0.0;
double t = 0.0;
System.out.println("step\tx\ty\ttime");
System.out.println("0\t0.0\t0.0\t0.0");
for (int i = 1; i <= steps; i++) {
    t += timeIncrement;
    x += xIncrement;
    y = yVelocity * t + 0.5 * ACCELERATION * t * t;
    System.out.println(i + "\t" + round2(x) + "\t" +
                       round2(y) + "\t" + round2(t));
}
```

Unstructured Solution

We can put all of these pieces together to form a complete program. Let's first look at an unstructured version that includes most of the code in `main`. This version also includes some new `println` statements at the beginning that give a brief introduction to the user:

```
1  // This program computes the trajectory of a projectile.
2
3  import java.util.*; // for Scanner
4
5  public class Projectile {
6      // constant for Earth acceleration in meters/second^2
7      public static final double ACCELERATION = -9.81;
8
9      public static void main(String[] args) {
10         Scanner console = new Scanner(System.in);
11
12         System.out.println("This program computes the");
13         System.out.println("trajectory of a projectile given");
14         System.out.println("its initial velocity and its");
15         System.out.println("angle relative to the");
16         System.out.println("horizontal.");
17         System.out.println();
18
19         System.out.print("velocity (meters/second)? ");
20         double velocity = console.nextDouble();
21         System.out.print("angle (degrees)? ");
22         double angle = Math.toRadians(console.nextDouble());
```

```
23          System.out.print("number of steps to display? ");
24          int steps = console.nextInt();
25          System.out.println();
26
27          double xVelocity = velocity * Math.cos(angle);
28          double yVelocity = velocity * Math.sin(angle);
29          double totalTime = -2.0 * yVelocity / ACCELERATION;
30          double timeIncrement = totalTime / steps;
31          double xIncrement = xVelocity * timeIncrement;
32
33          double x = 0.0;
34          double y = 0.0;
35          double t = 0.0;
36          System.out.println("step\tx\ty\ttime");
37          System.out.println("0\t0.0\t0.0\t0.0");
38          for (int i = 1; i <= steps; i++) {
39              t += timeIncrement;
40              x += xIncrement;
41              y = yVelocity * t + 0.5 * ACCELERATION * t * t;
42              System.out.println(i + "\t" + round2(x) + "\t" +
43                                 round2(y) + "\t" + round2(t));
44          }
45      }
46
47      public static double round2(double n) {
48          return Math.round(n * 100.0) / 100.0;
49      }
50  }
```

The following is a sample execution of the program:

```
This program computes the
trajectory of a projectile given
its initial velocity and its
angle relative to the
horizontal.

velocity (meters/second)? 30
angle (degrees)? 50
number of steps to display? 10

step  x      y      time
0     0.0    0.0    0.0
1     9.03   9.69   0.47
```

```
2     18.07   17.23   0.94
3     27.1    22.61   1.41
4     36.14   25.84   1.87
5     45.17   26.92   2.34
6     54.21   25.84   2.81
7     63.24   22.61   3.28
8     72.28   17.23   3.75
9     81.31   9.69    4.22
10    90.35   0.0     4.69
```

From the log of execution, you can see that the projectile reaches a maximum height of 26.92 meters after 2.34 seconds (the fifth step) and that it lands 90.35 meters from where it began after 4.69 seconds (the tenth step).

This version of the program works, but we don't generally want to include so much code in the `main` method. The next section explores how to break up the program into smaller pieces.

Structured Solution

There are three major blocks of code in the `main` method of the `Projectile` program: a series of `println` statements that introduce the program to the user, a series of statements that prompt the user for the three values used to produce the table, and then the code that produces the table itself.

So, in pseudocode, the overall structure looks like this:

```
give introduction.
prompt for velocity, angle, and number of steps.
produce table.
```

The first and third steps are easily turned into methods, but not the middle step. This step prompts the user for values that we need to produce the table. If we turned it into a method, it would have to somehow return three values back to `main`. A method can return only a single value, so unfortunately we can't turn this step into a method. We could turn it into three different methods, one for each of the three values, but each of those methods would be just two lines long, so it's not clear that doing so would improve the overall structure.

The main improvement we can make, then, is to split off the introduction and the table into separate methods. Another improvement we can make is to turn the physics displacement formula into its own method. It is always a good idea to turn equations into methods. Introducing those methods, we get the following structured version of the program:

```
1  // This program computes the trajectory of a projectile.
2
3  import java.util.*;     // for Scanner
4
5  public class Projectile2 {
```

```
6       // constant for Earth acceleration in meters/second^2
7       public static final double ACCELERATION = -9.81;
8
9       public static void main(String[] args) {
10          Scanner console = new Scanner(System.in);
11          giveIntro();
12
13          System.out.print("velocity (meters/second)? ");
14          double velocity = console.nextDouble();
15          System.out.print("angle (degrees)? ");
16          double angle = Math.toRadians(console.nextDouble());
17          System.out.print("number of steps to display? ");
18          int steps = console.nextInt();
19          System.out.println();
20
21          printTable(velocity, angle, steps);
22      }
23
24      // prints a table showing the trajectory of an object given
25      // its initial velocity and angle and including the given
26      // number of steps in the table
27      public static void printTable(double velocity,
28                                    double angle, int steps) {
29          double xVelocity = velocity * Math.cos(angle);
30          double yVelocity = velocity * Math.sin(angle);
31          double totalTime = -2.0 * yVelocity / ACCELERATION;
32          double timeIncrement = totalTime / steps;
33          double xIncrement = xVelocity * timeIncrement;
34
35          double x = 0.0;
36          double y = 0.0;
37          double t = 0.0;
38          System.out.println("step\tx\ty\ttime");
39          System.out.println("0\t0.0\t0.0\t0.0");
40          for (int i = 1; i <= steps; i++) {
41              t += timeIncrement;
42              x += xIncrement;
43              y = displacement(yVelocity, t, ACCELERATION);
44              System.out.println(i + "\t" + round2(x) + "\t" +
45                                 round2(y) + "\t" + round2(t));
46          }
47      }
48
49      // gives a brief introduction to the user
```

```
50      public static void giveIntro() {
51          System.out.println("This program computes the");
52          System.out.println("trajectory of a projectile given");
53          System.out.println("its initial velocity and its");
54          System.out.println("angle relative to the");
55          System.out.println("horizontal.");
56          System.out.println();
57      }
58
59      // returns the vertical displacement for a body given
60      // initial velocity v, elapsed time t, and acceleration a
61      public static double displacement(double v, double t,
62                                        double a) {
63          return v * t + 0.5 * a * t * t;
64      }
65
66      // rounds n to 2 digits after the decimal point
67      public static double round2(double n) {
68          return Math.round(n * 100.0) / 100.0;
69      }
70  }
```

This version executes the same way as the earlier version.

Chapter Summary

Methods may be written to accept parameters, which are sets of characteristics that distinguish different members of a family of tasks. Parameters allow data values to flow into a method, which can change the way the method executes. A method declared with a set of parameters can perform an entire family of similar tasks instead of exactly one task.

When primitive values such as those of type `int` or `double` are passed as parameters, their values are copied into the method. Primitive parameters send values into a method but not out of it; the method can use the data values but cannot affect the value of any variables outside it.

Two methods can have the same name if they declare different parameters. This is called overloading.

Methods can be written to return values to the calling code. This feature allows a method to perform a complex computation and then provide its result back to the calling code. The type of the return value must be declared in the method's header and is called the method's return type.

Java has a class called `Math` that contains several useful static methods that you can use in your programs, such as powers, square roots, and logarithms.

An object is an entity that combines data and operations. Some objects in Java include `Strings`, which are sequences of text characters, and `Scanners`, which read user input.

Objects contain methods that implement their behavior. To call a method on an object, write its name, followed by a dot, followed by the method name.

A `String` object holds a sequence of characters. The characters have indexes, starting with 0 for the first character.

An exception is an error that occurs when a program has performed an illegal action and is unable to continue executing normally.

Some programs are interactive and respond to input from the user. These programs should print a message to the user, also called a prompt, asking for the input.

Java has a class called `Scanner` that reads input from the keyboard. A `Scanner` can read various pieces of input (also called tokens) from an input source. It can read either one token at a time or an entire line at a time.

Self-Check Problems

Section 3.1: Parameters

1. Which of the following is the correct syntax for a method header with parameters?

   a. `public static void example(x, y) {`
   b. `public static (int x, int y) example() {`
   c. `public static void example(int x,y) {`
   d. `public static void example(x: int, y: int) {`
   e. `public static void example(int x, int y) (`

2. What output is produced by the following program?

```
1  public class MysteryNums {
2      public static void main(String[] args) {
3          int x = 15;
4          sentence(x, 42);
5
6          int y = x - 5;
7          sentence(y, x + y);
8      }
9
10     public static void sentence(int num1, int num2) {
11         System.out.println(num1 + " " + num2);
12     }
13 }
```

3. The following program has 9 mistakes. What are they?

```
1  public class Oops3 {
2      public static void main() {
3          double bubble = 867.5309;
4          double x = 10.01;
5          printer(double x, double y);
```

```
6           printer(x);
7           printer("barack", "obama");
8           System.out.println("z = " + z);
9       }
10
11      public static void printer(x, y double) {
12          int z = 5;
13          System.out.println("x = " + double x + " and y = " + y);
14          System.out.println("The value from main is: " + bubble);
15      }
16  }
```

4. What output is produced by the following program?

```
1   public class Odds {
2       public static void main(String[] args) {
3           printOdds(3);
4           printOdds(17 / 2);
5
6           int x = 25;
7           printOdds(37 - x + 1);
8       }
9
10      public static void printOdds(int n) {
11          for (int i = 1; i <= n; i++) {
12              int odd = 2 * i - 1;
13              System.out.print(odd + " ");
14          }
15          System.out.println();
16      }
17  }
```

5. What is the output of the following program?

```
1   public class Weird {
2       public static void main(String[] args) {
3           int number = 8;
4           halfTheFun(11);
5           halfTheFun(2 - 3 + 2 * 8);
6           halfTheFun(number);
7           System.out.println("number = " + number);
8       }
9
10      public static void halfTheFun(int number) {
11          number = number / 2;
12          for (int count = 1; count <= number; count++) {
13              System.out.print(count + " ");
```

```
14          }
15          System.out.println();
16      }
17  }
```

6. What is the output of the following program?

```
1  public class MysteryNumbers {
2      public static void main(String[] args) {
3          String one = "two";
4          String two = "three";
5          String three = "1";
6          int number = 20;
7
8          sentence(one, two, 3);
9          sentence(two, three, 14);
10         sentence(three, three, number + 1);
11         sentence(three, two, 1);
12         sentence("eight", three, number / 2);
13     }
14
15     public static void sentence(String three, String one, int number) {
16         System.out.println(one + " times " + three + " = " + (number * 2));
17     }
18 }
```

7. What output is produced by the following program?

```
1  public class MysteryWho {
2      public static void main(String[] args) {
3          String whom = "her";
4          String who = "him";
5          String it = "who";
6          String he = "it";
7          String she = "whom";
8
9          sentence(he, she, it);
10         sentence(she, he, who);
11         sentence(who, she, who);
12         sentence(it, "stu", "boo");
13         sentence(it, whom, who);
14     }
15
16     public static void sentence(String she, String who, String whom) {
17         System.out.println(who + " and " + whom + " like " + she);
18     }
19 }
```

8. What output is produced by the following program?

```
1  public class MysteryTouch {
2      public static void main(String[] args) {
3          String head = "shoulders";
4          String knees = "toes";
5          String elbow = "head";
6          String eye = "eyes and ears";
7          String ear = "eye";
8
9          touch(ear, elbow);
10         touch(elbow, ear);
11         touch(head, "elbow");
12         touch(eye, eye);
13         touch(knees, "Toes");
14         touch(head, "knees " + knees);
15     }
16
17     public static void touch(String elbow, String ear) {
18         System.out.println("touch your " + elbow + " to your " + ear);
19     }
20 }
```

9. What output is produced by the following program?

```
1  public class MysterySoda {
2      public static void main(String[] args) {
3          String soda = "Coke";
4          String pop = "Pepsi";
5          String Coke = "pop";
6          String Pepsi = "soda";
7          String say = pop;
8
9          carbonated(Coke, soda, pop);
10         carbonated(pop, Pepsi, Pepsi);
11         carbonated("pop", pop, "Kool-Aid");
12         carbonated(say, "say", pop);
13     }
14     public static void carbonated(String Coke, String soda, String pop) {
15         System.out.println("say " + soda + " not " + pop + " or " + Coke);
16     }
17 }
```

10. Write a method called `printStrings` that accepts a `String` and a number of repetitions as parameters and prints that `String` the given number of times with a space after each time. For example, the call

```
printStrings("abc", 5);
```

will print the following output:

```
abc abc abc abc abc
```

11. The `System.out.println` command works on many different types of values, such as integers or `double`s. What is the term for such a method?

Section 3.2: Methods That Return Values

12. What is wrong with the following program?

```
1  public class Temperature {
2      public static void main(String[] args) {
3          double tempf = 98.6;
4          double tempc = 0.0;
5          ftoc(tempf, tempc);
6          System.out.println("Body temp in C is: " + tempc);
7      }
8
9      // converts Fahrenheit temperatures to Celsius
10     public static void ftoc(double tempf, double tempc) {
11         tempc = (tempf - 32) * 5 / 9;
12     }
13 }
```

13. Evaluate the following expressions:

a. `Math.abs(-1.6)`
b. `Math.abs(2 + -4)`
c. `Math.pow(6, 2)`
d. `Math.pow(5 / 2, 6)`
e. `Math.ceil(9.1)`
f. `Math.ceil(115.8)`
g. `Math.max(7, 4)`
h. `Math.min(8, 3 + 2)`
i. `Math.min(-2, -5)`
j. `Math.sqrt(64)`
k. `Math.sqrt(76 + 45)`
l. `100 + Math.log10(100)`
m. `13 + Math.abs(-7) - Math.pow(2, 3) + 5`
n. `Math.sqrt(16) * Math.max(Math.abs(-5), Math.abs(-3))`
o. `7 - 2 + Math.log10(1000) + Math.log(Math.pow(Math.E, 5))`
p. `Math.max(18 - 5, Math.ceil(4.6 * 3))`

14. What output is produced by the following program?

```
1  public class MysteryReturn {
2      public static void main(String[] args) {
3          int x = 1, y = 2, z = 3;
4          z = mystery(x, z, y);
5          System.out.println(x + " " + y + " " + z);
```

```
6           x = mystery(z, z, x);
7           System.out.println(x + " " + y + " " + z);
8           y = mystery(y, y, z);
9           System.out.println(x + " " + y + " " + z);
10      }
11
12      public static int mystery(int z, int x, int y) {
13          z--;
14          x = 2 * y + z;
15          y = x - 1;
16          System.out.println(y + " " + z);
17          return x;
18      }
19  }
```

15. Write the result of each expression. Note that a variable's value changes only if you reassign it using the = operator.

```
double grade = 2.7;
Math.round(grade);                                  // grade =
grade = Math.round(grade);                          // grade =

double min = Math.min(grade, Math.floor(2.9));      //   min =

double x = Math.pow(2, 4);                          //     x =
x = Math.sqrt(64);                                  //     x =

int count = 25;
Math.sqrt(count);                                   // count =
count = (int) Math.sqrt(count);                     // count =

int a = Math.abs(Math.min(-1, -3));                 //     a =
```

16. Write a method called `min` that takes three integers as parameters and returns the smallest of the three values; for example, a call of `min(3, -2, 7)` would return `-2`, and a call of `min(19, 27, 6)` would return `6`. Use `Math.min` to write your solution.

17. Write a method called `countQuarters` that takes an `int` representing a number of cents as a parameter and returns the number of quarter coins represented by that many cents. Don't count any whole dollars, because those would be dispensed as dollar bills. For example, `countQuarters(64)` would return 2, because 64 cents is equivalent to 2 quarters with 14 cents left over. A call of `countQuarters(1278)` would return 3, because after the 12 dollars are taken out, 3 quarters remain in the 78 cents that are left.

Section 3.3: Using Objects

18. What output is produced by the following code?

```
String first = "James";
String last = "Kirk";
String middle = "T."
System.out.println(last);
System.out.println("My name is " + first);
System.out.println(first + " " + last);
```

```
System.out.println(last + ", " + first + " " + middle);
System.out.println(middle + " is for Tiberius");
```

19. Assuming that the following variables have been declared:

```
//       index 0123456789012345
String str1 = "Frodo Baggins";
String str2 = "Gandalf the GRAY";
```

evaluate the following expressions:

a. `str1.length()`
b. `str1.charAt(7)`
c. `str2.charAt(0)`
d. `str1.indexOf("o")`
e. `str2.toUpperCase()`
f. `str1.toLowerCase().indexOf("B")`
g. `str1.substring(4)`
h. `str2.substring(3, 14)`
i. `str2.replace("a", "oo")`
j. `str2.replace("gray", "white")`
k. `"str1".replace("r", "range")`

20. Assuming that the following variables have been declared:

```
String str1 = "Q.E.D.";
String str2 = "Arcturan Megadonkey";
String str3 = "Sirius Cybernetics Corporation";
```

evaluate the following expressions:

a. `str1.length()`
b. `str2.length()`
c. `str1.toLowerCase()`
d. `str2.toUpperCase()`
e. `str1.substring(2, 4)`
f. `str2.substring(10, 14)`
g. `str1.indexOf("D")`
h. `str1.indexOf(".")`
i. `str2.indexOf("donkey")`
j. `str3.indexOf("X")`
k. `str2 + str3.charAt(17)`
l. `str3.substring(9, str3.indexOf("e"))`
m. `str3.substring(7, 12)`
n. `str2.toLowerCase().substring(9, 13) + str3.substring(18, str3.length() - 7)`

21. Consider the following `String`:

```
String quote = "Four score and seven years ago";
```

What expression produces the new `String "SCORE"`? What expression produces `"four years"`?

22. In a program reading user input by means of a `Scanner`, the user types the following line of input:

```
Hello there. 1+2 is 3 and 1.5 squared is 2.25!
```

Into how many tokens will this line be split? What is each token?

23. Consider the following code fragment:

```
Scanner console = new Scanner(System.in);
System.out.print("How much money do you have? ");
double money = console.nextDouble();
```

Describe what will happen when the user types each of the following values. If the code will run successfully, describe the value that will be stored in the variable `money`.

a. `34.50`
b. `6`
c. `$25.00`
d. `million`
e. `100*5`
f. `600x000`
g. `none`
h. `645`

24. Write Java code to read an integer from the user, then print that number multiplied by 2. You may assume that the user types a valid integer.

25. Consider the following program. Modify the code to use a `Scanner` to prompt the user for the values of `low` and `high`.

```
1  public class SumNumbers {
2      public static void main(String[] args) {
3          int low = 1;
4          int high = 1000;
5          int sum = 0;
6          for (int i = low; i <= high; i++) {
7              sum += i;
8          }
9          System.out.println("sum = " + sum);
10     }
11 }
```

Below is a sample execution in which the user asks for the sum of the values 1 through 10:

```
low? 1
high? 10
sum = 55
```

26. Write Java code that prompts the user for a phrase and a number of times to repeat it, then prints the phrase the requested number of times. Here is an example dialogue with the user:

```
What is your phrase? His name is Robert Paulson.
How many times should I repeat it? 3
His name is Robert Paulson.
His name is Robert Paulson.
His name is Robert Paulson.
```

Exercises

1. Write a method called `printNumbers` that accepts a maximum number as an argument and prints each number from 1 up to that maximum, inclusive, boxed by square brackets. For example, consider the following calls:

```
printNumbers(15);
printNumbers(5);
```

These calls should produce the following output:

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

You may assume that the value passed to `printNumbers` is 1 or greater.

2. Write a method called `printPowersOf2` that accepts a maximum number as an argument and prints each power of 2 from 2^0 (1) up to that maximum power, inclusive. For example, consider the following calls:

```
printPowersOf2(3);
printPowersOf2(10);
```

These calls should produce the following output:

```
1 2 4 8
1 2 4 8 16 32 64 128 256 512 1024
```

You may assume that the value passed to `printPowersOf2` is 0 or greater. (The `Math` class may help you with this problem. If you use it, you may need to cast its results from `double` to `int` so that you don't see a `.0` after each number in your output. Also try to write this program without using the `Math` class.)

3. Write a method called `printPowersOfN` that accepts a base and an exponent as arguments and prints each power of the base from base^0 (1) up to that maximum power, inclusive. For example, consider the following calls:

```
printPowersOfN(4, 3);
printPowersOfN(5, 6);
printPowersOfN(-2, 8);
```

These calls should produce the following output:

```
1 4 16 64
1 5 25 125 625 3125 15625
1 -2 4 -8 16 -32 64 -128 256
```

You may assume that the exponent passed to `printPowersOfN` has a value of 0 or greater. (The `Math` class may help you with this problem. If you use it, you may need to cast its results from `double` to `int` so that you don't see a `.0` after each number in your output. Also try to write this program without using the `Math` class.)

4. Write a method called `printSquare` that accepts a minimum and maximum integer and prints a square of lines of increasing numbers. The first line should start with the minimum, and each line that follows should start with the next-higher number. The sequence of numbers on a line wraps back to the minimum after it hits the maximum. For example, the call

```
printSquare(3, 7);
```

should produce the following output:

```
34567
45673
```

```
56734
67345
73456
```

If the maximum passed is less than the minimum, the method produces no output.

5. Write a method called `printGrid` that accepts two integers representing a number of rows and columns and prints a grid of integers from 1 to (rows * columns) in column major order. For example, the call

```
printGrid(4, 6);
```

should produce the following output:

```
1 5 9 13 17 21
2 6 10 14 18 22
3 7 11 15 19 23
4 8 12 16 20 24
```

6. Write a method called `largerAbsVal` that takes two integers as parameters and returns the larger of the two absolute values. A call of `largerAbsVal(11, 2)` would return `11`, and a call of `largerAbsVal(4, -5)` would return `5`.

7. Write a variation of the `largestAbsVal` method from the last exercise that takes three integers as parameters and returns the largest of their three absolute values. For example, a call of `largestAbsVal(7, -2, -11)` would return `11`, and a call of `largestAbsVal(-4, 5, 2)` would return `5`.

8. Write a method called `quadratic` that solves quadratic equations and prints their roots. Recall that a quadratic equation is a polynomial equation in terms of a variable x of the form $ax^2 + bx + c = 0$. The formula for solving a quadratic equation is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here are some example equations and their roots:

$x^2 - 7x + 12$: $x = 4, x = 3$

$x^2 + 3x + 2$: $x = -2, x = -1$

Your method should accept the coefficients a, b, and c as parameters and should print the roots of the equation. You may assume that the equation has two real roots, though mathematically this is not always the case.

9. Write a method called `lastDigit` that returns the last digit of an integer. For example, `lastDigit(3572)` should return `2`. It should work for negative numbers as well. For example, `lastDigit(-947)` should return `7`.

10. Write a method called `area` that accepts as a parameter the radius of a circle and that returns the area of the circle. For example, the call `area(2.0)` should return `12.566370614359172`. Recall that area can be computed as pi (π) times the radius squared and that Java has a constant called `Math.PI`.

11. Write a method called `distance` that accepts four integer coordinates x_1, y_1, x_2, and y_2 as parameters and computes the distance between points (x_1, y_1) and (x_2, y_2) on the Cartesian plane. The equation for the distance is

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

For example, the call of `distance(1, 0, 4, 4)` would return `5.0` and the call of `distance(10, 2, 3, 15)` would return `14.7648230602334`.

12. Write a method called `scientific` that accepts a real number base and an exponent as parameters and computes the base times 10 to the exponent, as seen in scientific notation. For example, the call of `scientific(6.23, 5)` would return `623000.0` and the call of `scientific(1.9, -2)` would return `0.019`.

13. Write a method called `pay` that accepts two parameters: a real number for a TA's salary, and an integer for the number of hours the TA worked this week. The method should return how much money to pay the TA. For example, the call `pay(5.50, 6)` should return `33.0`. The TA should receive "overtime" pay of $1\frac{1}{2}$ times the normal salary for any hours above 8. For example, the call `pay(4.00, 11)` should return (4.00 * 8) + (6.00 * 3) or `50.0`.

14. Write a method called `cylinderSurfaceArea` that accepts a radius and height as parameters and returns the surface area of a cylinder with those dimensions. For example, the call `cylinderSurfaceArea(3.0, 4.5)` should return `141.3716694115407`. The formula for the surface area of a cylinder with radius r and height h is the following:

surface area $= 2\pi r^2 + 2\pi rh$

15. Write a method called `sphereVolume` that accepts a radius as a parameter and returns the volume of a sphere with that radius. For example, the call `sphereVolume(2.0)` should return `33.510321638291124`. The formula for the volume of a sphere with radius r is the following:

volume $= \frac{4}{3}\pi r^3$

16. Write a method called `triangleArea` that accepts the three side lengths of a triangle as parameters and returns the area of a triangle with those side lengths. For example, the call `triangleArea(8, 5.2, 7.1)` should return `18.151176098258745`. To compute the area, use Heron's formula, which states that the area of a triangle whose three sides have lengths a, b, and c, is the following. The formula is based on the computed value s, a length equal to half the perimeter of the triangle:

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

$$s = \frac{a+b+c}{2}$$

17. Write a method called `padString` that accepts two parameters: a string and an integer representing a length. The method should pad the parameter string with spaces until its length is the given length. For example, `padString("hello", 8)` should return `"hello   "`. (This sort of method is useful when trying to print output that lines up horizontally.) If the string's length is already at least as long as the length parameter, your method should return the original string. For example, `padString("congratulations", 10)` should return `"congratulations"`.

18. Write a method called `vertical` that accepts a string as its parameter and prints each letter of the string on separate lines. For example, a call of `vertical("hey now")` should produce the following output:

```
h
e
y

n
o
w
```

19. Write a method called `printReverse` that accepts a string as its parameter and prints the characters in opposite order. For example, a call of `printReverse("hello there!")` should print `"!ereht olleh"`. If the empty string is passed, the method should produce no output.

20. Write a method called `inputBirthday` that accepts a `Scanner` for the console as a parameter and prompts the user to enter a month, day, and year of birth, then prints the birthdate in a suitable format. Here is an example dialogue with the user:

```
On what day of the month were you born? 8
What is the name of the month in which you were born? May
During what year were you born? 1981
You were born on May 8, 1981. You're mighty old!
```

21. Write a method called `processName` that accepts a `Scanner` for the console as a parameter and prompts the user to enter a full name, then prints the name in reverse order (i.e., last name, first name). Here is an example dialogue with the user:

```
Please enter your full name: Sammy Jankis
Your name in reverse order is Jankis, Sammy
```

22. Write a complete program called `CollegeAdmit` with the behavior shown below. Read user input for a student's grade point average and SAT exam score. A GPA below 1.8 will cause the student to be rejected; an SAT score below 900 will also cause a rejection. Otherwise the student is accepted.

```
University admission program!
What is your GPA? 3.2
What is your SAT score? 1280
You were accepted!
```

Programming Projects

1. Write a program that produces images of Christmas trees as output. It should have a method with two parameters: one for the number of segments in the tree and one for the height of each segment. For example, the tree shown here on the left has three segments of height 4 and the one on the right has two segments of height 5:

```
   *          *
  ***        ***
 *****      *****
*******    *******
  ***      *********
 *****       ***
*******     *****
*********  *******
  *****    *********
 *******  ***********
*********     *
***********    *
    *       *******
    *
  *******
```

2. A certain bank offers 6.5% interest on savings accounts, compounded annually. Create a table that shows how much money a person will accumulate over a period of 25 years, assuming that the person makes an initial investment of $1000 and deposits $100 each year after the first. Your table should indicate for each year the current balance, the interest, the new deposit, and the new balance.

3. Write a program that shows the total number of presents that the person in the song "The Twelve Days of Christmas" received on each day, as indicated in Table 3.5.

Table 3.5 Twelve Days of Christmas

Day	Presents Received	Total Presents
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
. . .	. . .	. . .

4. Write a program that prompts for the lengths of the sides of a triangle and reports the three angles.

5. Write a program that computes the spherical distance between two points on the surface of the Earth, given their latitudes and longitudes. This is a useful operation because it tells you how far apart two cities are if you multiply the distance by the radius of the Earth, which is roughly 6372.795 km.

   Let φ_1, λ_1, and φ_2, λ_2 be the latitude and longitude of two points, respectively. $\Delta\lambda$, the longitudinal difference, and $\Delta\sigma$, the angular difference/distance in radians, can be determined as follows from the spherical law of cosines:

   $$\Delta\sigma = \arccos(\sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos\Delta\lambda)$$

   For example, consider the latitude and longitude of two major cities:

   - Nashville, TN: N 36°7.2′, W 86°40.2′
   - Los Angeles, CA: N 33°56.4′, W 118°24.0′

   You must convert these coordinates to radians before you can use them effectively in the formula. After conversion, the coordinates become

   - Nashville: $\varphi_1 = 36.12° = 0.6304$ rad, $\lambda_1 = -86.67° = -1.5127$ rad
   - Los Angeles: $\varphi_2 = 33.94° = 0.5924$ rad, $\lambda_2 = -118.40° = -2.0665$ rad

   Using these values in the angular distance equation, you get

   $$1\Delta\sigma = 6372.795 \times 0.45306 = 2887.259 \text{ km}$$

   Thus, the distance between these cities is about 2887 km, or 1794 miles. (Note: To solve this problem, you will need to use the `Math.acos` method, which returns an arccosine angle in radians.)

6. Write a program that produces calendars as output. Your program should have a method that outputs a single month's calendar like the one below, given parameters to specify how many days are in the month and what the date of the first Sunday is in that month. In the month shown below, these values are 31 and 6, respectively.

```
  Sun    Mon    Tue    Wed    Thu    Fri    Sat
+------+------+------+------+------+------+------+
|      |      |   1  |   2  |   3  |   4  |   5  |
|   6  |   7  |   8  |   9  |  10  |  11  |  12  |
|  13  |  14  |  15  |  16  |  17  |  18  |  19  |
|  20  |  21  |  22  |  23  |  24  |  25  |  26  |
|  27  |  28  |  29  |  30  |  31  |      |      |
+------+------+------+------+------+------+------+
```

   One tricky part of this program is making the various columns line up properly with proper widths. We will learn better ways of formatting output in the next chapter. For now, you may copy the following `helper` method into your program and call it to turn a number into a left-padded string of a given exact width. For example, the call `System.out.print(padded(7, 5))`, prints "   7" (the number 7 with four leading spaces).

```
// Returns a string of the number n, left-padded
// with spaces until it is at least the given width.
public static String padded(int n, int width) {
    String s = "" + n;
    for (int i = s.length(); i < width; i++) {
        s = " " + s;
    }
    return s;
}
```