CHAPTER 2

USING OBJECTS

CHAPTER GOALS

To learn about variables

To understand the concepts of classes and objects

To be able to call methods

To learn about arguments and return values

To be able to browse the API documentation

To implement test programs

To understand the difference between objects and object references

To write programs that display simple shapes

CHAPTER CONTENTS

Most useful programs don't just manipulate numbers and strings. Instead, they deal with data items that are more complex and that more closely represent entities in the real world. Examples of these data items include bank accounts, employee records, and graphical shapes.

The Java language is ideally suited for designing and manipulating such data items, or *objects*. In Java, you implement *classes* that describe the behavior of these objects. In this chapter, you will learn how to manipulate objects that belong to classes that have already been implemented. This will prepare you for the next chapter, in which you will learn how to implement your own classes.

2.1 Objects and Classes

When you write a computer program, you put it together from certain "building blocks". In Java, you build programs from *objects*. Each object has a particular behavior, and you can manipulate it to achieve certain effects.

As an analogy, think of a home builder who constructs a house from certain parts: doors, windows, walls, pipes, a furnace, a water heater, and so on. Each of these elements has a particular function, and they work together to fulfill a common purpose. Note that the home builder is not concerned with how to build a window or a water heater. These elements are readily available, and the builder's job is to integrate them into the house.

Each part that a home builder uses, such as a furnace or a water heater, fulfills a particular function. Similarly, you build programs from objects, each of which has a particular behavior.

Of course, computer programs are more abstract than houses, and the objects that make up a computer program aren't as tangible as a window or a water heater. But the analogy holds well: A programmer produces a working program from elements with the desired functionality—the objects. In this chapter, you will learn the basics about using objects written by other programmers.

2.1.1 Using Objects

Objects are entities in your program that you manipulate by calling methods.

An **object** is an entity that you can manipulate by calling one or more of its **methods**. A method consists of a sequence of instructions that can access the internal data of an object. When you call the method, you do not know exactly what those instructions are, or even how the object is organized internally. However, the behavior of the method is well defined, and that is what matters to us when we use it.

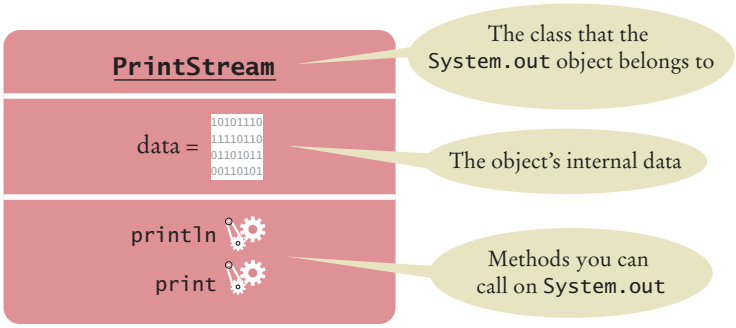


Figure 1 Representation of the `System.out` Object

A method is a sequence of instructions that accesses the data of an object.

For example, you saw in Chapter 1 that `System.out` refers to an object. You manipulate it by calling the `println` method. When the `println` method is called, some activities occur inside the object, and the ultimate effect is that text appears in the console window. You don't know how that happens, and that's OK. What matters is that the method carries out the work that you requested.

Figure 1 shows a representation of the `System.out` object. The internal data is symbolized by a sequence of zeroes and ones. Think of each method (symbolized by the gears) as a piece of machinery that carries out its assigned task.

In general, think of an object as an entity that can do work for you when you call its methods. How the work is done is not important to the programmer using the object.

In the remainder of this chapter, you will see other objects and the methods that they can carry out.

You can think of a water heater as an object that can carry out the "get hot water" method. When you call that method to enjoy a hot shower, you don't care whether the water heater uses gas or solar power.

2.1.2 Classes

In Chapter 1, you encountered two objects:

- `System.out`
- `"Hello, World!"`

A class describes a set of objects with the same behavior.

Each of these objects belongs to a different **class**. The `System.out` object belongs to the `PrintStream` class. The `"Hello, World!"` object belongs to the `String` class. Of course, there are many more `String` objects, such as `"Goodbye"` or `"Mississippi"`. They all have something in common—you can invoke the same methods on all strings. You will see some of these methods in Section 2.3.

As you will see in Chapter 11, you can construct objects of the `PrintStream` class other than `System.out`. Those objects write data to files or other destinations instead of the console. Still, all `PrintStream` objects share common behavior. You can invoke the `println` and `print` methods on any `PrintStream` object, and the printed values are sent to their destination.

Of course, the objects of the `PrintStream` class have a completely different behavior than the objects of the `String` class. You could not call `println` on a `String` object. A string wouldn't know how to send itself to a console window or file.

As you can see, different classes have different responsibilities. A string knows about the letters that it contains, but it does not know how to display them to a human or to save them to a file.

All objects of a `Window` class share the same behavior.

1. In Java, objects are grouped into classes according to their behavior. Would a window object and a water heater object belong to the same class or to different classes? Why?
2. Some light bulbs use a glowing filament, others use a fluorescent gas. If you consider a light bulb a Java object with an "illuminate" method, would you need to know which kind of bulb it is?

Practice It Now you can try these exercises at the end of the chapter: R2.1, R2.2.

2.2 Variables

Before we continue with the main topic of this chapter—the behavior of objects—we need to go over some basic programming terminology. In the following sections, you will learn about the concepts of variables, types, and assignment.

2.2.1 Variable Declarations

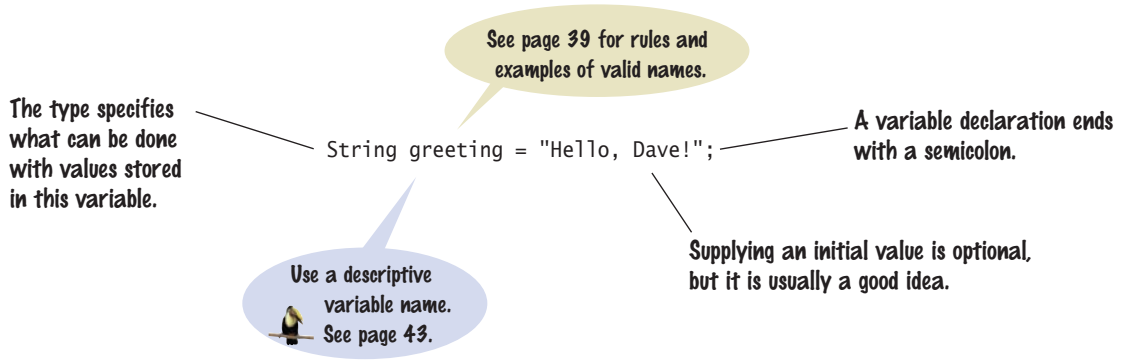
When your program manipulates objects, you will want to store the objects and the values that their methods return, so that you can use them later. In a Java program, you use variables to store values. The following statement declares a variable named `width`:

```
int width = 20;
```

Like a variable in a computer program, a parking space has an identifier and a contents.

Syntax 2.1 Variable Declaration

Syntax *typeName variableName = value*;
or
typeName variableName;



A variable is a storage location with a name.

When declaring a variable, you usually specify an initial value.

When declaring a variable, you also specify the type of its values.

A **variable** is a storage location in a computer program. Each variable has a name and holds a value.

A variable is similar to a parking space in a parking garage. The parking space has an identifier (such as "J 053"), and it can hold a vehicle. A variable has a name (such as `width`), and it can hold a value (such as 20). When declaring a variable, you usually want to **initialize** it. That is, you specify the value that should be stored in the variable. Consider again this variable declaration:

```
int width = 20;
```

The variable `width` is initialized with the value 20.

Like a parking space that is restricted to a certain type of vehicle (such as a compact car, motorcycle, or electric vehicle), a variable in Java stores data of a specific type. Java supports quite a few data types: numbers, text strings, files, dates, and many others. You must specify the type whenever you declare a variable (see Syntax 2.1).

The `width` variable is an **integer**, a whole number without a fractional part. In Java, this type is called `int`.

Note that the type comes before the variable name:

```
int width = 20;
```

After you have declared and initialized a variable, you can use it. For example,

```
int width = 20;
System.out.println(width);
int area = width * width;
```

Table 1 shows several examples of variable declarations.

Each parking space is suitable for a particular type of vehicle, just as each variable holds a value of a particular type.

Table 1 Variable Declarations in Java

Variable Name	Comment
`int width = 20;`	Declares an integer variable and initializes it with 20.
`int perimeter = 4 * width;`	The initial value need not be a fixed value. (Of course, `width` must have been previously declared.)
`String greeting = "Hi!";`	This variable has the type `String` and is initialized with the string "Hi".
`height = 30;`	**Error:** The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.2.5.
`int width = "20";`	**Error:** You cannot initialize a number with the string "20". (Note the quotation marks.)
`int width;`	Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 42.
`int width, height;`	Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.

2.2.2 Types

Use the `int` type for numbers that cannot have a fractional part.

In Java, there are several different types of numbers. You use the `int` type to denote a whole number without a fractional part. For example, suppose you count the number of cars in a parking lot. The counter must be an integer number—you cannot have a fraction of a car.

When a fractional part is required (such as in the number 22.5), we use **floating-point numbers**. The most commonly used type for floating-point numbers in Java is called `double`. Here is the declaration of a floating-point variable:

```
double milesPerGallon = 22.5;
```

Use the `double` type for floating-point numbers.

You can combine numbers with the `+` and `-` operators, as in `width + 10` or `width - 1`. To multiply two numbers, use the `*` operator. For example, 2 × *width* is written as `2 * width`. Use the `/` operator for division, such as `width / 2`.

Numbers can be combined by arithmetic operators such as +, -, and *.

As in mathematics, the `*` and `/` operator bind more strongly than the `+` and `-` operators. That is, `width + height * 2` means the sum of `width` and the product `height * 2`. If you want to multiply the sum by 2, use parentheses: `(width + height) * 2`.

Not all types are number types. For example, the value `"Hello"` has the type `String`. You need to specify that type when you define a variable that holds a string:

```
String greeting = "Hello";
```

A type specifies the operations that can be carried out with its values.

Types are important because they indicate what you can do with a variable. For example, consider the variable `width`. It's type is `int`. Therefore, you can multiply the value that it holds with another number. But the type of `greeting` is `String`. You can't multiply a string with another number. (You will see in Section 2.3.1 what you can do with strings.)

2.2.3 Names

When you declare a variable, you should pick a name that explains its purpose. For example, it is better to use a descriptive name, such as `milesPerGallon`, than a terse name, such as `mpg`.

In Java, there are a few simple rules for the names of variables, methods, and classes:

1. Names must start with a letter or the underscore (_) character, and the remaining characters must be letters, numbers, or underscores. (Technically, the `$` symbol is allowed as well, but you should not use it—it is intended for names that are automatically generated by tools.)
2. You cannot use other symbols such as `?` or `%`. Spaces are not permitted inside names either. You can use uppercase letters to denote word boundaries, as in `milesPerGallon`. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.)

3. Names are **case sensitive**, that is, `milesPerGallon` and `milespergallon` are different names.
4. You cannot use **reserved words** such as `double` or `class` as names; these words are reserved exclusively for their special Java meanings. (See Appendix C for a listing of all reserved words in Java.)

By convention, variable names should start with a lowercase letter.

It is a convention among Java programmers that names of variables and methods start with a lowercase letter (such as `milesPerGallon`). Class names should start with an uppercase letter (such as `HelloPrinter`). That way, it is easy to tell them apart.

Table 2 shows examples of legal and illegal variable names in Java.

Table 2 Variable Names in Java

Variable Name	Comment
`distance_1`	Names consist of letters, numbers, and the underscore character.
`x`	In mathematics, you use short variable names such as x or y. This is legal in Java, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 43).
⚠ `CanVolume`	**Caution:** Names are case sensitive. This variable name is different from `canVolume`, and it violates the convention that variable names should start with a lowercase letter.
🚫 `6pack`	**Error:** Names cannot start with a number.
🚫 `can volume`	**Error:** Names cannot contain spaces.
🚫 `double`	**Error:** You cannot use a reserved word as a name.
🚫 `miles/gal`	**Error:** You cannot use symbols such as / in names.

2.2.4 Comments

As your programs get more complex, you should add **comments**, explanations for human readers of your code. For example, here is a comment that explains the value used to initialize a variable:

```
double milesPerGallon = 33.8; // The average fuel efficiency of new U.S. cars in 2011
```

This comment explains the significance of the value 33.8 to a human reader. The compiler does not process comments at all. It ignores everything from a // delimiter to the end of the line.

Use comments to add explanations for humans who read your code. The compiler ignores comments.

It is a good practice to provide comments. This helps programmers who read your code understand your intent. In addition, you will find comments helpful when you review your own programs.

You use the // delimiter for short comments. If you have a longer comment, enclose it between /* and */ delimiters. The compiler ignores these delimiters and everything in between. For example,

```
/*
   In most countries, fuel efficiency is measured in liters per hundred
   kilometer. Perhaps that is more useful—it tells you how much gas you need
   to purchase to drive a given distance. Here is the conversion formula.
*/
double fuelEfficiency = 235.214583 / milesPerGallon;
```

2.2.5 Assignment

Use the assignment operator (=) to change the value of a variable.

You can change the value of a variable with the assignment operator (=). For example, consider the variable declaration

```
int width = 10; 1
```

If you want to change the value of the variable, simply assign the new value:

```
width = 20; 2
```

The assignment replaces the original value of the variable (see Figure 2).

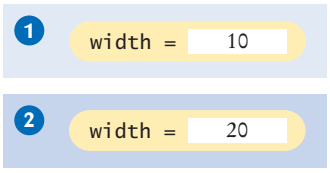


Figure 2
Assigning a New Value to a Variable

It is an error to use a variable that has never had a value assigned to it. For example, the following assignment statement has an error:

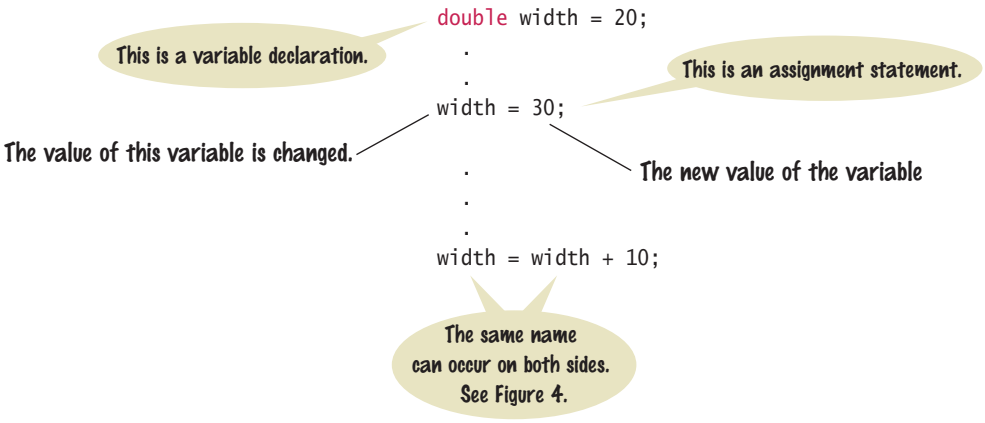
```
int height;
int width = height;    // ERROR—uninitialized variable height
```

The compiler will complain about an "**uninitialized variable**" when you use a variable that has never been assigned a value. (See Figure 3.)

Figure 3
An Uninitialized Variable

Syntax 2.2 Assignment

Syntax *variableName* = *value*;



All variables must be initialized before you access them.

The remedy is to assign a value to the variable before you use it:

```
int height = 20;
int width = height; // OK
```

The right-hand side of the = symbol can be a mathematical expression. For example,

```
width = height + 10;
```

This means "compute the value of `height + 10` and store that value in the variable `width`".

The assignment operator = does *not* denote mathematical equality.

In the Java programming language, the = operator denotes an *action*, namely to replace the value of a variable. This usage differs from the mathematical usage of the = symbol as a statement about equality. For example, in Java, the following statement is entirely legal:

```
width = width + 10;
```

This means "compute the value of `width + 10` (1) and store that value in the variable `width` (2)" (see Figure 4).

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a program that demonstrates variables and assignments.

In Java, it is not a problem that the variable `width` is used on both sides of the = symbol. Of course, in mathematics, the equation *width* = *width* + 10 has no solution.

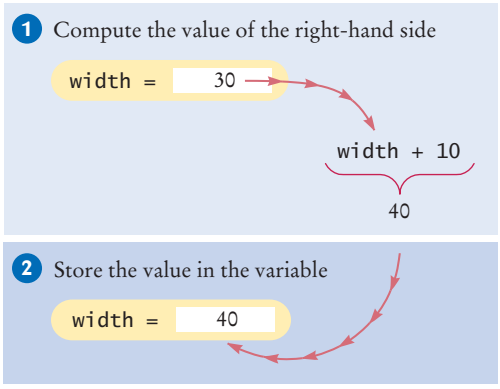


Figure 4
Executing the Statement `width = width + 10`

3. What is wrong with the following variable declaration?

```
int miles per gallon = 39.4
```

4. Declare and initialize two variables, `unitPrice` and `quantity`, to contain the unit price of a single item and the number of items purchased. Use reasonable initial values.
5. Use the variables declared in Self Check 4 to display the total purchase price.
6. What are the types of the values `0` and `"0"`?
7. Which number type would you use for storing the area of a circle?
8. Which of the following are legal identifiers?

```
Greeting1
g
void
101dalmatians
Hello, World
<greeting>
```

9. Declare a variable to hold your name. Use camel case in the variable name.
10. Is `12 = 12` a valid expression in the Java language?
11. How do you change the value of the `greeting` variable to `"Hello, Nina!"`?
12. How would you explain assignment using the parking space analogy?

Practice It Now you can try these exercises at the end of the chapter: R2.3, R2.4, R2.6.

Common Error 2.1

Using Undeclared or Uninitialized Variables

You must declare a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
int perimeter = 4 * width; // ERROR: width not yet declared
int width = 20;
```

In your program, the statements are compiled in order. When the compiler reaches the first statement, it does not know that `width` will be declared in the next line, and it reports an error. The remedy is to reorder the declarations so that each variable is declared before it is used.

A related error is to leave a variable uninitialized:

```
int width;
int perimeter = 4 * width; // ERROR: width not yet initialized
```

The Java compiler will complain that you are using a variable that has not yet been given a value. The remedy is to assign a value to the variable before it is used.

Common Error 2.2

Confusing Variable Declarations and Assignment Statements

Suppose your program declares a variable as follows:

```
int width = 20;
```

If you want to change the value of the variable, you use an assignment statement:

```
width = 30;
```

It is a common error to accidentally use another variable declaration:

```
int width = 30; // ERROR—starts with int and is therefore a declaration
```

But there is already a variable named `width`. The compiler will complain that you are trying to declare another variable with the same name.

Programming Tip 2.1

Choose Descriptive Variable Names

In algebra, variable names are usually just one letter long, such as p or A, maybe with a subscript such as p_1. You might be tempted to save yourself a lot of typing by using short variable names in your Java programs:

```
int a = w * h;
```

Compare that statement with the following one:

```
int area = width * height;
```

The advantage is obvious. Reading `width` is much easier than reading `w` and then figuring out that it must mean "width".

In practical programming, descriptive variable names are particularly important when programs are written by more than one person. It may be obvious to you that `w` stands for width, but is it obvious to the person who needs to update your code years later? For that matter, will you yourself remember what `w` means when you look at the code a month from now?

2.3 Calling Methods

A program performs useful work by calling methods on its objects. In this section, we examine how to supply values in a method, and how to obtain the result of the method.

2.3.1 The Public Interface of a Class

You use an object by calling its methods. All objects of a given class share a common set of methods. For example, the `PrintStream` class provides methods for its objects (such as `println` and `print`). Similarly, the `String` class provides methods that you can apply to `String` objects. One of them is the `length` method. The `length` method counts the number of characters in a string. You can apply that method to any object of type `String`. For example, the sequence of statements:

```
String greeting = "Hello, World!";
int numberOfCharacters = greeting.length();
```

sets `numberOfCharacters` to the length of the `String` object `"Hello, World!"`. After the instructions in the `length` method are executed, `numberOfCharacters` is set to 13. (The quotation marks are not part of the string, and the `length` method does not count them.)

When calling the `length` method, you do not supply any values inside the parentheses. Also note that the `length` method does not produce any visible output. It returns a value that is subsequently used in the program.

Let's look at another method of the `String` class. When you apply the `toUpperCase` method to a `String` object, the method creates another `String` object that contains the characters of the original string, with lowercase letters converted to uppercase. For example, the sequence of statements

```
String river = "Mississippi";
String bigRiver = river.toUpperCase();
```

sets `bigRiver` to the `String` object `"MISSISSIPPI"`.

The public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.

The `String` class declares many other methods besides the `length` and `toUpperCase` methods—you will learn about many of them in Chapter 4. Collectively, the methods form the **public interface** of the class, telling you what you can do with the objects of the class. A class also declares a *private implementation*, describing the data inside its objects and the instructions for its methods. Those details are hidden from the programmers who use objects and call methods.

The controls of a car form its public interface. The private implementation is under the hood.

Figure 5 shows two objects of the `String` class. Each object stores its own data (drawn as boxes that contain characters). Both objects support the same set of methods—the public interface that is specified by the `String` class.

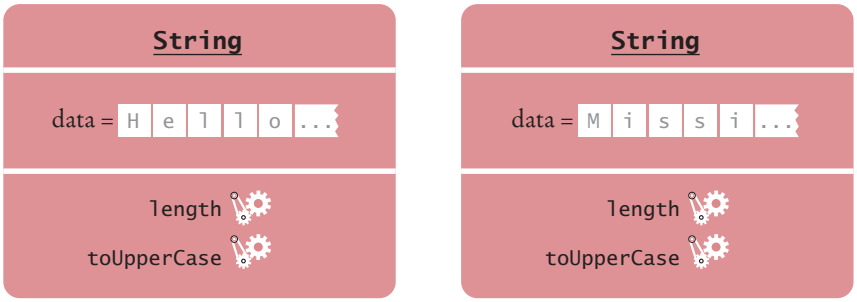


Figure 5 A Representation of Two `String` Objects

2.3.2 Method Arguments

An argument is a value that is supplied in a method call.

Most methods require values that give details about the work that the method needs to do. For example, when you call the `println` method, you must supply the string that should be printed. Computer scientists use the technical term **argument** for method inputs. We say that the string `greeting` is an argument of the method call

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

Figure 6 illustrates passing the argument to the method.

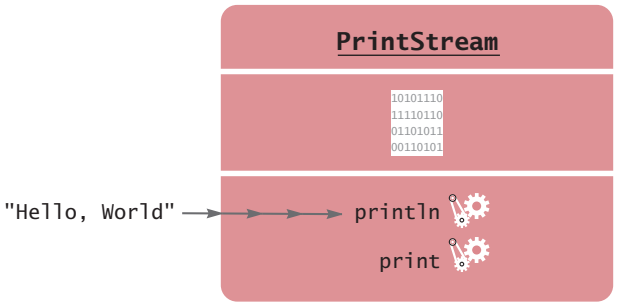


Figure 6 Passing an Argument to the `println` Method

At this tailor shop, the customer's measurements and the fabric are the arguments of the `sew` *method. The return value is the finished garment.*

Some methods require multiple arguments; others don't require any arguments at all. An example of the latter is the `length` method of the `String` class (see Figure 7). All the information that the `length` method requires to do its job—namely, the character sequence of the string—is stored in the object that carries out the method.

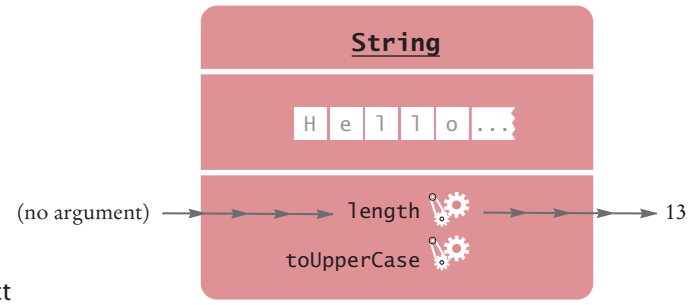


Figure 7
Invoking the `length` Method on a `String` Object

2.3.3 Return Values

The return value of a method is a result that the method has computed.

Some methods, such as the `println` method, carry out an action for you. Other methods compute and return a value. For example, the `length` method returns a value, namely the number of characters in the string. You can store the return value in a variable:

```
int numberOfCharacters = greeting.length();
```

You can also use the return value of one method as an argument of another method:

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

The method call `greeting.length()` returns a value—the integer 13. The return value becomes an argument of the `println` method. Figure 8 shows the process.

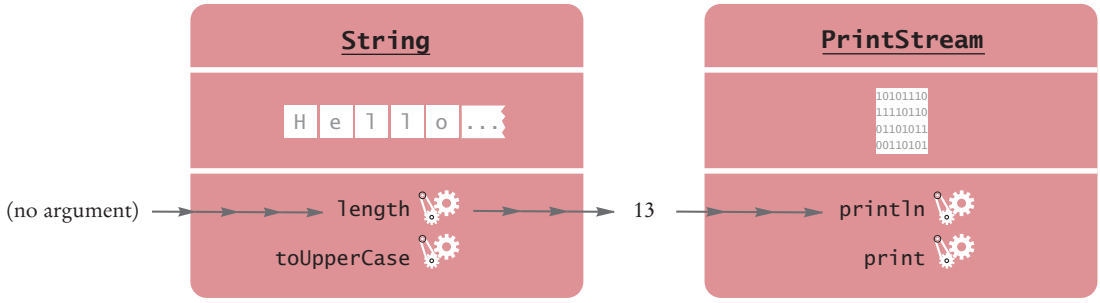


Figure 8 Passing the Result of a Method Call to Another Method

Not all methods return values. One example is the `println` method. The `println` method interacts with the operating system, causing characters to appear in a window. But it does not return a value to the code that calls it.

Let us analyze a more complex method call. Here, we will call the `replace` method of the `String` class. The `replace` method carries out a search-and-replace operation, similar to that of a word processor. For example, the call

```
river.replace("issipp", "our")
```

constructs a new string that is obtained by replacing all occurrences of `"issipp"` in `"Mississippi"` with `"our"`. (In this situation, there was only one replacement.) The method returns the `String` object `"Missouri"`. You can save that string in a variable:

```
river = river.replace("issipp", "our");
```

Or you can pass it to another method:

```
System.out.println(river.replace("issipp", "our"));
```

As Figure 9 shows, this method call

- Is invoked on a `String` object: `"Mississippi"`
- Has two arguments: the strings `"issipp"` and `"our"`
- Returns a value: the string `"Missouri"`

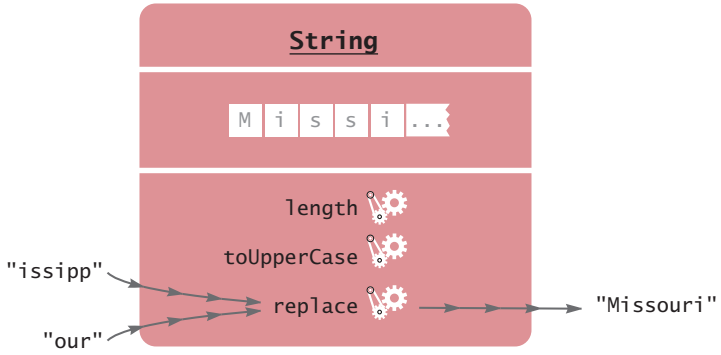


Figure 9 Calling the `replace` Method

Table 3 Method Arguments and Return Values

Example	Comments
`System.out.println(greeting)`	`greeting` is an argument of the `println` method.
`greeting.replace("e","3")`	The `replace` method has two arguments, in this case `"e"` and `"3"`.
`greeting.length()`	The `length` method has no arguments.
`int n = greeting.length();`	The `length` method returns an integer value.
`System.out.println(n);`	The `println` method returns no value. In the API documentation, its return type is `void`.
`System.out.println(greeting.length());`	The return value of one method can become the argument of another.

2.3.4 Method Declarations

When a method is declared in a class, the declaration specifies the types of the arguments and the return value. For example, the `String` class declares the `length` method as

```
public int length()
```

That is, there are no arguments, and the return value has the type `int`. (For now, all the methods that we consider will be "public" methods—see Chapter 9 for more restricted methods.)

The `replace` method is declared as

```
public String replace(String target, String replacement)
```

To call the `replace` method, you supply two arguments, `target` and `replacement`, which both have type `String`. The returned value is another string.

When a method returns no value, the return type is declared with the reserved word **`void`**. For example, the `PrintStream` class declares the `println` method as

```
public void println(String output)
```

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a program that demonstrates method calls.

Occasionally, a class declares two methods with the same name and different argument types. For example, the `PrintStream` class declares a second method, also called `println`, as

```
public void println(int output)
```

That method is used to print an integer value. We say that the `println` name is **overloaded** because it refers to more than one method.

SELF CHECK

13. How can you compute the length of the string `"Mississippi"`?
14. How can you print out the uppercase version of `"Hello, World!"`?
15. Is it legal to call `river.println()`? Why or why not?
16. What are the arguments in the method call `river.replace("p", "s")`?
17. What is the result of the call `river.replace("p", "s")`?
18. What is the result of the call `greeting.replace("World", "Dave").length()`?
19. How is the `toUpperCase` method declared in the `String` class?

Practice It Now you can try these exercises at the end of the chapter: R2.7, R2.8, R2.9.

Programming Tip 2.2

Learn By Trying

When you learn about a new method, write a small program to try it out. For example, you can go right now to your Java development environment and run this program:

```
public class ReplaceDemo
{
   public static void main(String[] args)
   {
      String river = "Mississippi";
      System.out.println(river.replace("issipp", "our"));
   }
}
```

Then you can see with your own eyes what the `replace` method does. Also, you can run experiments. Does `replace` change every match, or only the first one? Try it out:

```
System.out.println(river.replace("i", "x"));
```

Set up your work environment to make this kind of experimentation easy and natural. Keep a file with the blank outline of a Java program around, so you can copy and paste it when needed. Alternatively, some development environments will automatically type the class and `main` method. Find out if yours does. Some environments even let you type commands into a window and show you the result right away, without having to make a `main` method to call `System.out.println` (see Figure 10).

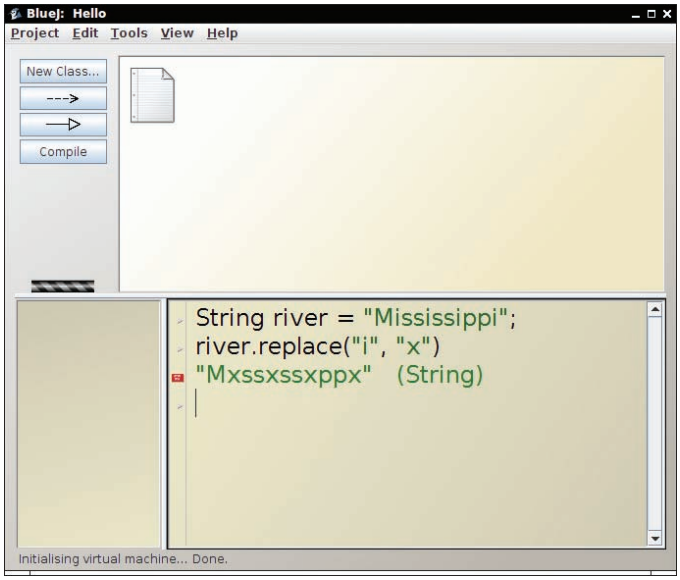


Figure 10 The Code Pad in BlueJ

2.4 Constructing Objects

Generally, when you want to use objects in your program, you need to specify their initial properties by *constructing* them.

To learn about object construction, we need to go beyond `String` objects and the `System.out` object. Let us turn to another class in the Java library: the `Rectangle` class. Objects of type `Rectangle` describe rectangular shapes. These objects are useful for a variety of purposes. You can assemble rectangles into bar charts, and you can program simple games by moving rectangles inside a window.

Note that a `Rectangle` object isn't a rectangular shape—it's an object that contains a set of numbers. The numbers *describe* the rectangle (see Figure 11). Each rectangle is described by the x- and y-coordinates of its top-left corner, its width, and its height.

Objects of the `Rectangle` *class describe rectangular shapes.*

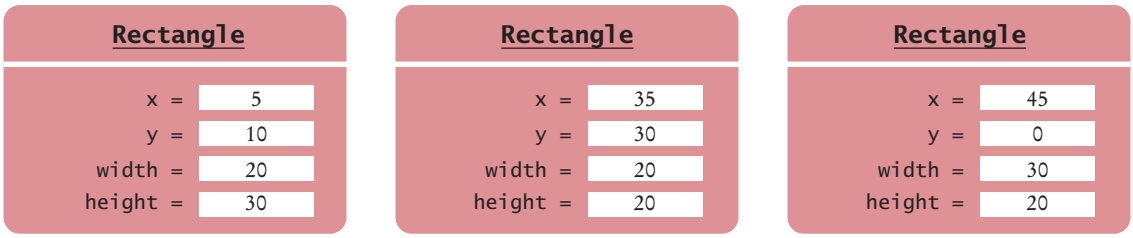


Figure 11 `Rectangle` Objects

It is very important that you understand this distinction. In the computer, a `Rectangle` object is a block of memory that holds four numbers, for example $x = 5$, $y = 10$, *width* = 20, *height* = 30. In the imagination of the programmer who uses a `Rectangle` object, the object describes a geometric figure.

Use the `new` operator, followed by a class name and arguments, to construct new objects.

To make a new rectangle, you need to specify the x, y, *width*, and *height* values. Then *invoke the* `new` *operator*, specifying the name of the class and the argument(s) required for constructing a new object. For example, you can make a new rectangle with its top-left corner at (5, 10), width 20, and height 30 as follows:

```
new Rectangle(5, 10, 20, 30)
```

Here is what happens in detail:

1. The `new` operator makes a `Rectangle` object.
2. It uses the arguments (in this case, 5, 10, 20, and 30) to initialize the object's data.
3. It returns the object.

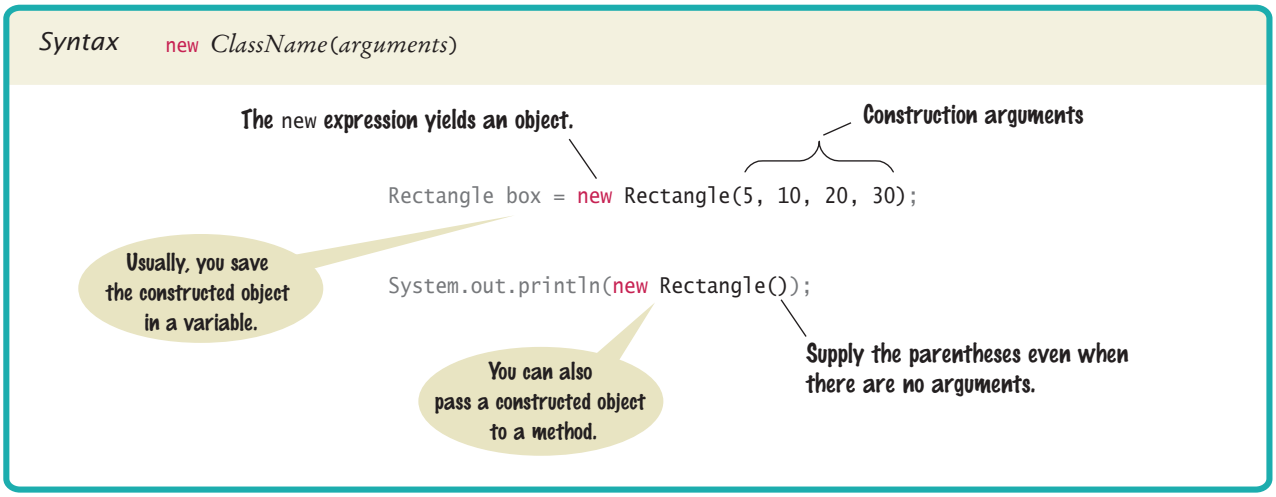
The process of creating a new object is called **construction**. The four values 5, 10, 20, and 30 are called the *construction arguments*.

The `new` expression yields an object, and you need to store the object if you want to use it later. Usually you assign the output of the `new` operator to a variable. For example,

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Syntax 2.3 Object Construction

Syntax `new` *ClassName*(*arguments*)



FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program that demonstrates constructors.

Some classes let you construct objects in multiple ways. For example, you can also obtain a `Rectangle` object by supplying no construction arguments at all (but you must still supply the parentheses):

```
new Rectangle()
```

This expression constructs a (rather useless) rectangle with its top-left corner at the origin (0, 0), width 0, and height 0.

20. How do you construct a square with center (100, 100) and side length 20?

21. Initialize the variables `box` and `box2` with two rectangles that touch each other.

22. The `getWidth` method returns the width of a `Rectangle` object. What does the following statement print?

```
System.out.println(new Rectangle().getWidth());
```

23. The `PrintStream` class has a constructor whose argument is the name of a file. How do you construct a `PrintStream` object with the construction argument `"output.txt"`?

24. Write a statement to save the object that you constructed in Self Check 23 in a variable.

Practice It Now you can try these exercises at the end of the chapter: R2.11, R2.14, R2.16.

Common Error 2.3

Trying to Invoke a Constructor Like a Method

Constructors are not methods. You can only use a constructor with the `new` operator, not to reinitialize an existing object:

```
box.Rectangle(20, 35, 20, 30); // Error—can't reinitialize object
```

The remedy is simple: Make a new object and overwrite the current one stored by `box`.

```
box = new Rectangle(20, 35, 20, 30); // OK
```

2.5 Accessor and Mutator Methods

An accessor method does not change the internal data of the object on which it is invoked. A mutator method changes the data.

In this section we introduce a useful terminology for the methods of a class. A method that accesses an object and returns some information about it, without changing the object, is called an **accessor method**. In contrast, a method whose purpose is to modify the internal data of an object is called a **mutator method**.

For example, the `length` method of the `String` class is an accessor method. It returns information about a string, namely its length. But it doesn't modify the string at all when counting the characters.

The `Rectangle` class has a number of accessor methods. The `getX`, `getY`, `getWidth`, and `getHeight` methods return the x- and y-coordinates of the top-left corner, the width, and the height values. For example,

```
double width = box.getWidth();
```

Now let us consider a mutator method. Programs that manipulate rectangles frequently need to move them around, for example, to display animations. The `Rectangle` class has a method for that purpose, called `translate`. (Mathematicians use the term "translation" for a rigid motion of the plane.) This method moves a rectangle by a certain distance in the x- and y-directions. The method call,

```
box.translate(15, 25);
```

moves the rectangle by 15 units in the x-direction and 25 units in the y-direction (see Figure 12). Moving a rectangle doesn't change its width or height, but it changes the top-left corner. Afterward, the rectangle that had its top-left corner at (5, 10) now has it at (20, 35).

This method is a mutator because it modifies the object on which the method is invoked.

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a program that demonstrates accessors and mutators.

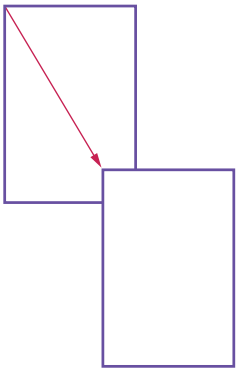


Figure 12 Using the `translate` Method to Move a Rectangle

SELF CHECK

25. What does this sequence of statements print?

```
Rectangle box = new Rectangle(5, 10, 20, 30);
System.out.println("Before: " + box.getX());
box.translate(25, 40);
System.out.println("After: " + box.getX());
```

26. What does this sequence of statements print?

```
Rectangle box = new Rectangle(5, 10, 20, 30);
System.out.println("Before: " + box.getWidth());
box.translate(25, 40);
System.out.println("After: " + box.getWidth());
```

27. What does this sequence of statements print?

```
String greeting = "Hello";
System.out.println(greeting.toUpperCase());
System.out.println(greeting);
```

28. Is the `toUpperCase` method of the `String` class an accessor or a mutator?

29. Which call to `translate` is needed to move the rectangle declared by `Rectangle box = new Rectangle(5, 10, 20, 30)` so that its top-left corner is the origin (0, 0)?

Practice It Now you can try these exercises at the end of the chapter: R2.17, E2.6, E2.8.

2.6 The API Documentation

The API (Application Programming Interface) documentation lists the classes and methods of the Java library.

The classes and methods of the Java library are listed in the **API documentation**. The API is the "application programming interface". A programmer who uses the Java classes to put together a computer program (or *application*) is an *application programmer*. That's you. In contrast, the programmers who designed and implemented the library classes such as `PrintStream` and `Rectangle` are *system programmers*.

You can find the API documentation on the Web. Point your web browser to `http://docs.oracle.com/javase/7/docs/api/index.html`. An abbreviated version of the API documentation is provided in Appendix D that may be easier to use at first, but you should eventually move on to the real thing.

2.6.1 Browsing the API Documentation

The API documentation documents all classes in the Java library—there are thousands of them (see Figure 13, top). Most of the classes are rather specialized, and only a few are of interest to the beginning programmer.

Locate the `Rectangle` link in the left pane, preferably by using the search function of your browser. Click on the link, and the right pane shows all the features of the `Rectangle` class (see Figure 13, bottom).

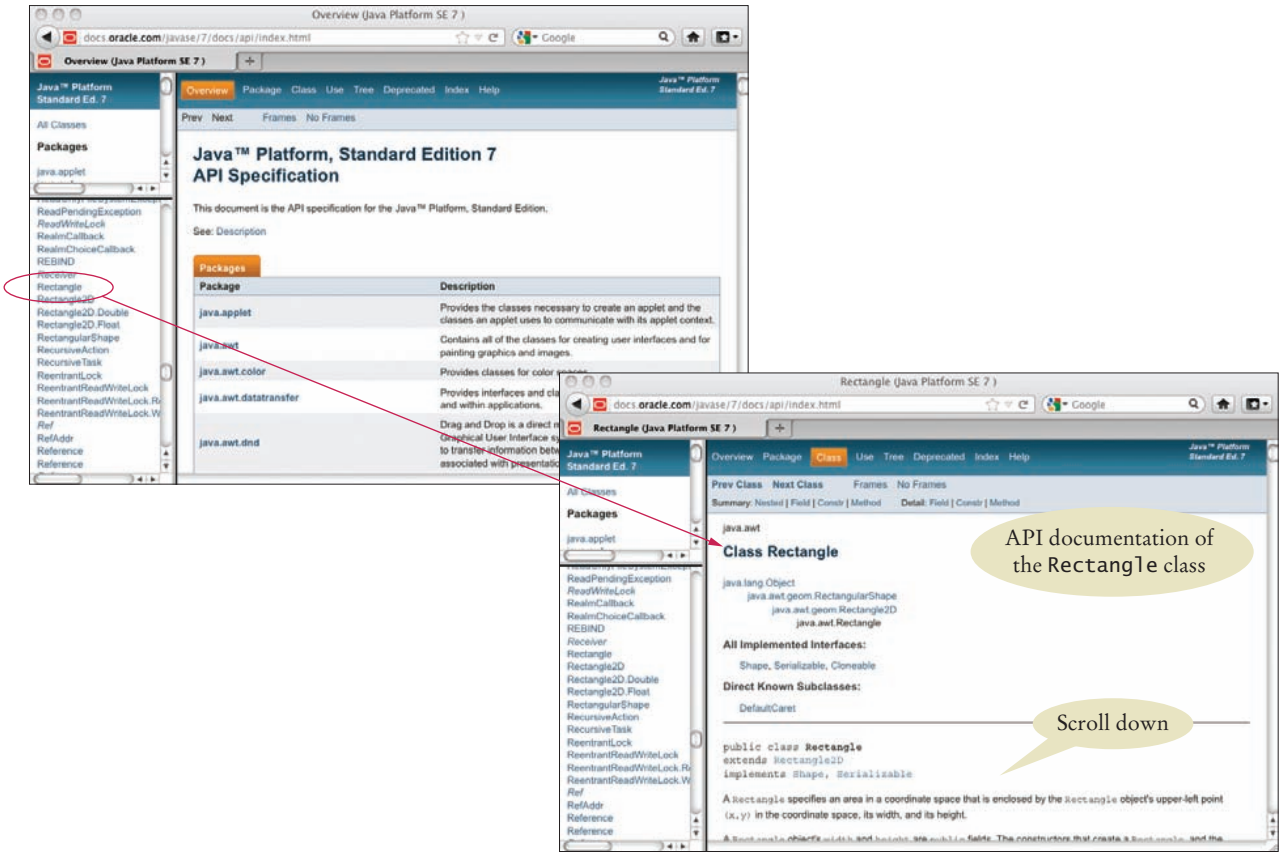


Figure 13 The API Documentation of the Standard Java Library

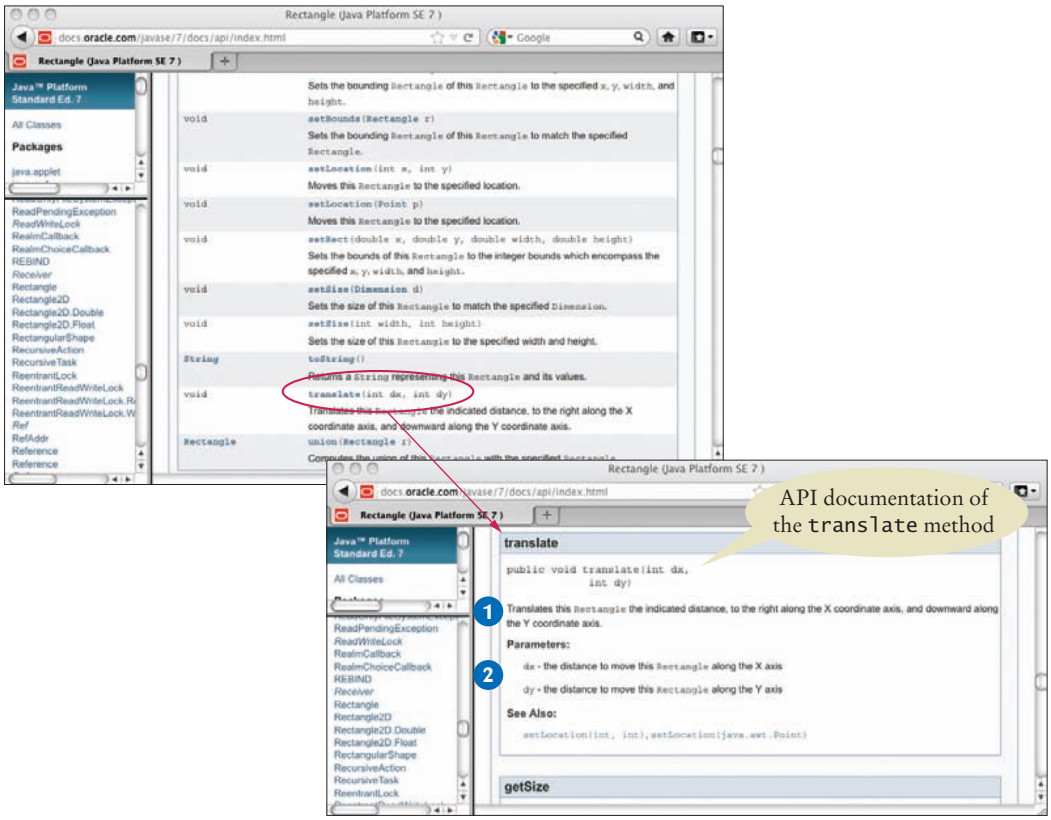


Figure 14 The Method Summary for the `Rectangle` Class

The API documentation for each class starts out with a section that describes the purpose of the class. Then come summary tables for the constructors and methods (see Figure 14, top). Click on a method's link to get a detailed description (see Figure 14, bottom).

The detailed description of a method shows

- The action that the method carries out. 1
- The types and names of the parameter variables that receive the arguments when the method is called. 2
- The value that it returns (or the reserved word `void` if the method doesn't return any value).

As you can see, the `Rectangle` class has quite a few methods. While occasionally intimidating for the beginning programmer, this is a strength of the standard library. If you ever need to do a computation involving rectangles, chances are that there is a method that does all the work for you.

For example, suppose you want to change the width or height of a rectangle. If you browse through the API documentation, you will find a `setSize` method with the description "Sets the size of this `Rectangle` to the specified width and height." The method has two arguments, described as

- `width` - the new width for this `Rectangle`
- `height` - the new height for this `Rectangle`

We can use this information to change the box object so that it is a square of side length 40. The name of the method is setSize, and we supply two arguments: the new width and height:

```
box.setSize(40, 40);
```

2.6.2 Packages

The API documentation contains another important piece of information about each class. The classes in the standard library are organized into **packages**. A package is a collection of classes with a related purpose. The Rectangle class belongs to the package java.awt (where awt is an abbreviation for "Abstract Windowing Toolkit"), which contains many classes for drawing windows and graphical shapes. You can see the package name java.awt in Figure 13, just above the class name.

Java classes are grouped into packages. Use the import statement to use classes that are declared in other packages.

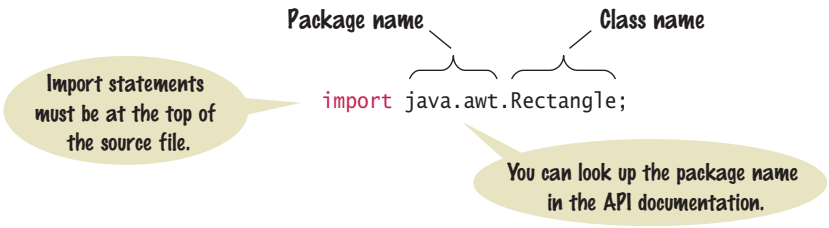
To use the Rectangle class from the java.awt package, you must *import* the package. Simply place the following line at the top of your program:

```
import java.awt.Rectangle;
```

Why don't you have to import the System and String classes? Because the System and String classes are in the java.lang package, and all classes from this package are automatically imported, so you never need to import them yourself.

Syntax 2.4 Importing a Class from a Package

Syntax `import` *packageName.ClassName*;



30. Look at the API documentation of the String class. Which method would you use to obtain the string "hello, world!" from the string "Hello, World!"?
31. In the API documentation of the String class, look at the description of the trim method. What is the result of applying trim to the string " Hello, Space ! "? (Note the spaces in the string.)
32. Look into the API documentation of the Rectangle class. What is the difference between the methods void translate(int x, int y) and void setLocation(int x, int y)?
33. The Random class is declared in the java.util package. What do you need to do in order to use that class in your program?

34. In which package is the `BigInteger` class located? Look it up in the API documentation.

Practice It Now you can try these exercises at the end of the chapter: R2.18, E2.4, E2.11.

Programming Tip 2.3

Don't Memorize—Use Online Help

The Java library has thousands of classes and methods. It is neither necessary nor useful trying to memorize them. Instead, you should become familiar with using the API documentation. Because you will need to use the API documentation all the time, it is best to download and install it onto your computer, particularly if your computer is not always connected to the Internet. You can download the documentation from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

2.7 Implementing a Test Program

A test program verifies that methods behave as expected.

In this section, we discuss the steps that are necessary to implement a test program. The purpose of a test program is to verify that one or more methods have been implemented correctly. A test program calls methods and checks that they return the expected results. Writing test programs is a very important skill.

In this section, we will develop a simple program that tests a method in the `Rectangle` class using these steps:

1. Provide a tester class.
2. Supply a `main` method.
3. Inside the `main` method, construct one or more objects.
4. Apply methods to the objects.
5. Display the results of the method calls.
6. Display the values that you expect to get.

Our sample test program tests the behavior of the `translate` method. Here are the key steps (which have been placed inside the `main` method of the `MoveTester` class).

```
Rectangle box = new Rectangle(5, 10, 20, 30);

// Move the rectangle
box.translate(15, 25);

// Print information about the moved rectangle
System.out.print("x: ");
System.out.println(box.getX());
System.out.println("Expected: 20");
```

We print the value that is returned by the `getX` method, and then we print a message that describes the value we expect to see.

Determining the expected result in advance is an important part of testing.

This is a very important step. You want to spend some time thinking about the expected result before you run a test program. This thought process will help you understand how your program should behave, and it can help you track down errors at an early stage. Finding and fixing errors early is a very effective strategy that can save you a great deal of time.

In our case, the rectangle has been constructed with the top-left corner at (5, 10). The x-direction is moved by 15, so we expect an x-value of 5 + 15 = 20 after the move.

Here is the program that tests the moving of a rectangle:

section_7/MoveTester.java

```
1  import java.awt.Rectangle;
2
3  public class MoveTester
4  {
5     public static void main(String[] args)
6     {
7        Rectangle box = new Rectangle(5, 10, 20, 30);
8
9        // Move the rectangle
10       box.translate(15, 25);
11
12       // Print information about the moved rectangle
13       System.out.print("x: ");
14       System.out.println(box.getX());
15       System.out.println("Expected: 20");
16
17       System.out.print("y: ");
18       System.out.println(box.getY());
19       System.out.println("Expected: 35");
20    }
21 }
```

Program Run

```
x: 20
Expected: 20
y: 35
Expected: 35
```

35. Suppose we had called `box.translate(25, 15)` instead of `box.translate(15, 25)`. What are the expected outputs?

36. Why doesn't the `MoveTester` program need to print the width and height of the rectangle?

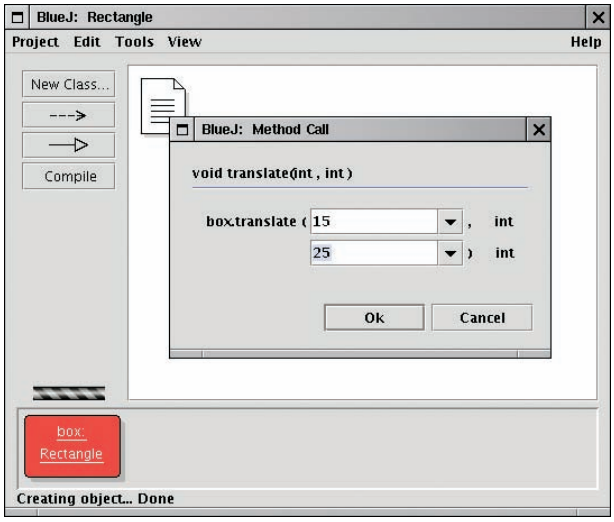
Practice It Now you can try these exercises at the end of the chapter: E2.1, E2.7, E2.13.

Special Topic 2.1

Testing Classes in an Interactive Environment

Some development environments are specifically designed to help students explore objects without having to provide tester classes. These environments can be very helpful for gaining insight into the behavior of objects, and for promoting object-oriented thinking. The BlueJ environment (shown in the figure) displays objects as blobs on a workbench.

You can construct new objects, put them on the workbench, invoke methods, and see the return values, all without writing a line of code. You can download BlueJ at no charge from `www.bluej.org`. Another excellent environment for interactively exploring objects is Dr. Java at `drjava.sourceforge.net`.



Testing a Method Call in BlueJ

WORKED EXAMPLE 2.1 **How Many Days Have You Been Alive?**

Explore the API of a class `Day` that represents a calendar day. Using that class, learn to write a program that computes how many days have elapsed since the day you were born. Go to `wiley.com/go/javaexamples` and download Worked Example 2.1.

WORKED EXAMPLE 2.2 **Working with Pictures**

Learn how to use the API of a `Picture` class to edit photos. Go to `wiley.com/go/javaexamples` and download Worked Example 2.2.

2.8 Object References

In Java, an object variable (that is, a variable whose type is a class) does not actually hold an object. It merely holds the *memory location* of an object. The object itself is stored elsewhere—see Figure 15.

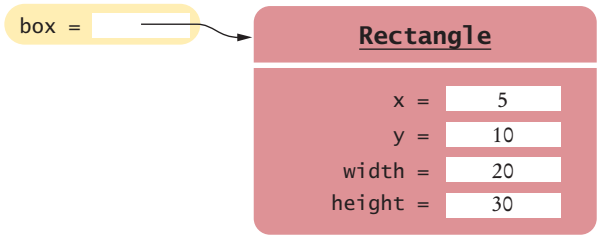


Figure 15 An Object Variable Containing an Object Reference

There is a reason for this behavior. Objects can be very large. It is more efficient to store only the memory location instead of the entire object.

An object reference describes the location of an object.

We use the technical term **object reference** to denote the memory location of an object. When a variable contains the memory location of an object, we say that it *refers* to an object. For example, after the statement

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

the variable `box` refers to the `Rectangle` object that the `new` operator constructed. Technically speaking, the `new` operator returned a reference to the new object, and that reference is stored in the `box` variable.

It is very important that you remember that the `box` variable *does not contain* the object. It *refers* to the object. Two object variables can refer to the same object:

```
Rectangle box2 = box;
```

Multiple object variables can contain references to the same object.

Now you can access the same `Rectangle` object as `box` and as `box2`, as shown in Figure 16.

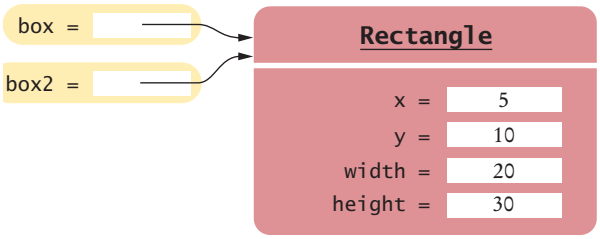


Figure 16 Two Object Variables Referring to the Same Object

In Java, numbers are not objects. Number variables actually store numbers. When you declare

```
int luckyNumber = 13;
```

then the `luckyNumber` variable holds the number 13, not a reference to the number (see Figure 17). The reason is again efficiency. Because numbers require little storage, it is more efficient to store them directly in a variable.

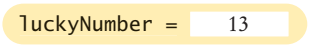


Figure 17 A Number Variable Stores a Number

Number variables store numbers. Object variables store references.

You can see the difference between number variables and object variables when you make a copy of a variable. When you copy a number, the original and the copy of the number are independent values. But when you copy an object reference, both the original and the copy are references to the same object.

Consider the following code, which copies a number and then changes the copy (see Figure 18):

```
int luckyNumber = 13; 1
int luckyNumber2 = luckyNumber; 2
luckyNumber2 = 12; 3
```

Now the variable `luckyNumber` contains the value 13, and `luckyNumber2` contains 12.

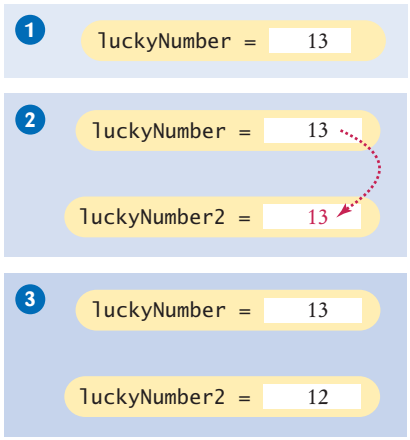


Figure 18 Copying Numbers

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a program that demonstrates the difference between copying numbers and object references.

Now consider the seemingly analogous code with `Rectangle` objects (see Figure 19).

```
Rectangle box = new Rectangle(5, 10, 20, 30); 1
Rectangle box2 = box; 2
box2.translate(15, 25); 3
```

Because `box` and `box2` refer to the same rectangle after step 2, both variables refer to the moved rectangle after the call to the `translate` method.

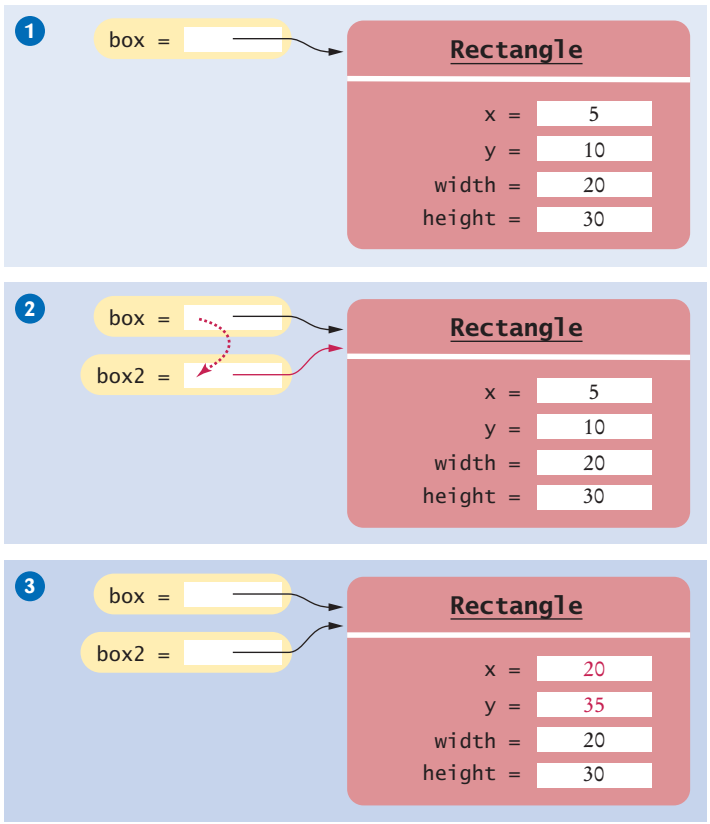


Figure 19 Copying Object References

You need not worry too much about the difference between objects and object references. Much of the time, you will have the correct intuition when you think of the "object `box`" rather than the technically more accurate "object reference stored in variable `box`". The difference between objects and object references only becomes apparent when you have multiple variables that refer to the same object.

SELF CHECK

37. What is the effect of the assignment `String greeting2 = greeting`?

38. After calling `greeting2.toUpperCase()`, what are the contents of `greeting` and `greeting2`?

Practice It Now you can try these exercises at the end of the chapter: R2.15, R2.19.

Computing & Society 2.1 Computer Monopoly

When International Business Machines Corporation (IBM), a successful manufacturer of punched-card equipment for tabulating data, first turned its attention to designing computers in the early 1950s, its planners assumed that there was a market for perhaps 50 such devices, for installation by the government, the military, and a few of the country's largest corporations. Instead, they sold about 1,500 machines of their System 650 model and went on to build and sell more powerful computers.

These computers, called mainframes, were huge. They filled rooms, which had to be climate-controlled to protect the delicate equipment. IBM was not the first company to build mainframe computers; that honor belongs to the Univac Corporation. However, IBM soon became the major player, partially because of its technical excellence and attention to customer needs and partially because it exploited its strengths and structured its products and services in a way that made it difficult for customers to mix them with those of other vendors.

As all of IBM's competitors fell on hard times, the U.S. government brought an antitrust suit against IBM in 1969. In the United States, it is legal to be a monopoly supplier, but it is not legal to use one's monopoly in one market to gain supremacy in another. IBM was accused of forcing customers to buy bundles of computers, software, and peripherals, making it impossible for other vendors of software and peripherals to compete.

The suit went to trial in 1975 and dragged on until 1982, when it was abandoned, largely because new waves of smaller computers had made it irrelevant.

In fact, when IBM offered its first personal computers, its operating system was supplied by an outside vendor, Microsoft, which became so dominant that it too was sued by the U.S. goverment for abusing its monopoly position in 1998. Microsoft was accused of bundling its web browser with its operating system. At the time, Microsoft allegedly threatened hardware makers that they would not receive a Windows license if they distributed the competing Netscape browser. In 2000, the company was found guilty of antitrust violations, and the judge ordered it broken up into an operating systems unit and an applications unit. The breakup was reversed on appeal, and a settlement in 2001 was largely unsuccessful in establishing alternatives for desktop software.

Now the computing landscape is shifting once again, toward mobile devices and cloud computing. As you observe that change, you may well see new monopolies in the making. When a software vendor needs the permission of a hardware vendor in order to place a product into an "app store", or when a maker of a digital book reader tries to coerce publishers into a particular pricing structure, the question arises whether such conduct is illegal exploitation of a monopoly position.

A Mainframe Computer

2.9 Graphical Applications

The following optional sections teach you how to write *graphical applications:* applications that display drawings inside a window. The drawings are made up of shape objects: rectangles, ellipses, and lines. The shape objects provide another source of examples, and many students enjoy the visual feedback.

2.9.1 Frame Windows

To show a frame, construct a `JFrame` object, set its size, and make it visible.

A graphical application shows information inside a **frame**: a window with a title bar, as shown in Figure 20. In this section, you will learn how to display a frame. In Section 2.9.2, you will learn how to create a drawing inside the frame.

A graphical application shows information inside a frame.

To show a frame, carry out the following steps:

1. Construct an object of the `JFrame` class:

   ```
   JFrame frame = new JFrame();
   ```

2. Set the size of the frame:

   ```
   frame.setSize(300, 400);
   ```

   This frame will be 300 pixels wide and 400 pixels tall. If you omit this step the frame will be 0 by 0 pixels, and you won't be able to see it. (Pixels are the tiny dots from which digital images are composed.)

3. If you'd like, set the title of the frame:

   ```
   frame.setTitle("An empty frame");
   ```

   If you omit this step, the title bar is simply left blank.

4. Set the "default close operation":

   ```
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   ```

   When the user closes the frame, the program automatically exits. Don't omit this step. If you do, the program keeps running even after the frame is closed.

5. Make the frame visible:

   ```
   frame.setVisible(true);
   ```

The simple program below shows all of these steps. It produces the empty frame shown in Figure 20.

The `JFrame` class is a part of the `javax.swing` package. Swing is the nickname for the graphical user interface library in Java. The "x" in `javax` denotes the fact that Swing started out as a Java *extension* before it was added to the standard library.

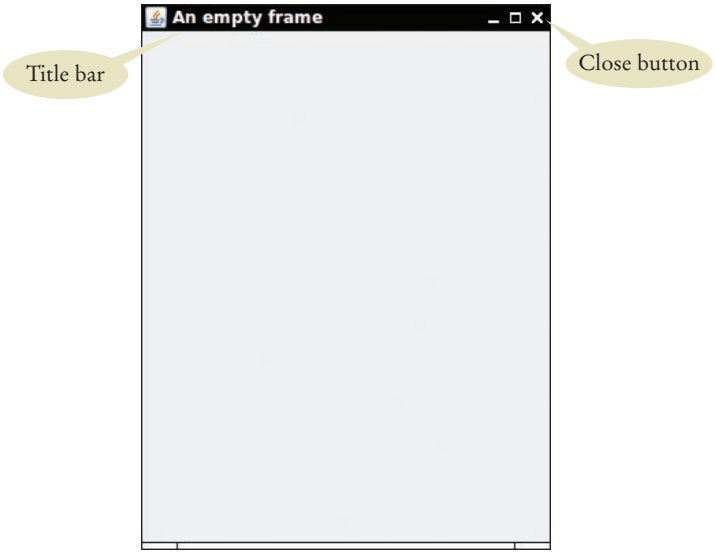


Figure 20 A Frame Window

We will go into much greater detail about Swing programming in Chapters 3, 10, and 19. For now, consider this program to be the essential plumbing that is required to show a frame.

section_9_1/EmptyFrameViewer.java

```
1  import javax.swing.JFrame;
2
3  public class EmptyFrameViewer
4  {
5     public static void main(String[] args)
6     {
7        JFrame frame = new JFrame();
8        frame.setSize(300, 400);
9        frame.setTitle("An empty frame");
10       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       frame.setVisible(true);
12    }
13 }
```

2.9.2 Drawing on a Component

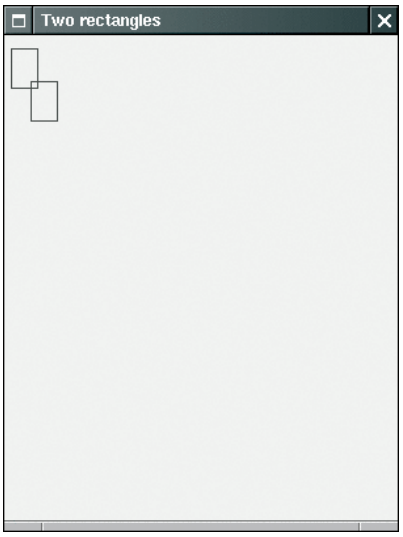
In this section, you will learn how to make shapes appear inside a frame window. The first drawing will be exceedingly modest: just two rectangles (see Figure 21). You'll soon see how to produce more interesting drawings. The purpose of this example is to show you the basic outline of a program that creates a drawing.

You cannot draw directly onto a frame. Instead, drawing happens in a **component** object. In the Swing toolkit, the `JComponent` class represents a blank component.

In order to display a drawing in a frame, declare a class that extends the `JComponent` class.

Because we don't want to add a blank component, we have to modify the `JComponent` class and specify how the component should be painted. The solution is to declare a new class that extends the `JComponent` class. You will learn about the process of extending classes in Chapter 9.

Figure 21
Drawing Rectangles



For now, simply use the following code as a template:

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Drawing instructions
   }
}
```

Place drawing instructions inside the `paintComponent` method. That method is called whenever the component needs to be repainted.

The `extends` reserved word indicates that our component class, `RectangleComponent`, can be used like a `JComponent`. However, the `RectangleComponent` class will be different from the plain `JComponent` class in one respect: Its `paintComponent` method will contain instructions to draw the rectangles.

When the component is shown for the first time, the `paintComponent` method is called automatically. The method is also called when the window is resized, or when it is shown again after it was hidden.

Use a cast to recover the `Graphics2D` object from the `Graphics` argument of the `paintComponent` method.

The `paintComponent` method receives an object of type `Graphics` as its argument. The `Graphics` object stores the graphics state—the current color, font, and so on—that are used for drawing operations. However, the `Graphics` class is not very useful. When programmers clamored for a more object-oriented approach to drawing graphics, the designers of Java created the `Graphics2D` class, which extends the `Graphics` class. Whenever the Swing toolkit calls the `paintComponent` method, it actually passes an object of type `Graphics2D` as the argument. Because we want to use the more sophisticated methods to draw two-dimensional graphics objects, we need to use the `Graphics2D` class. This is accomplished by using a **cast**:

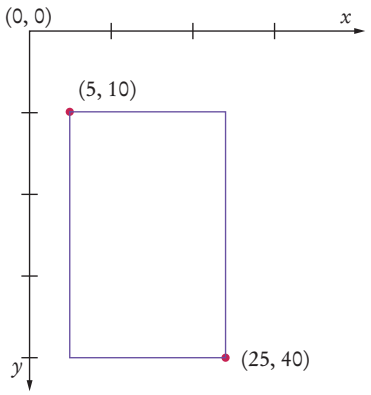
```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Recover Graphics2D
      Graphics2D g2 = (Graphics2D) g;
      . . .
   }
}
```

Chapter 9 has more information about casting. For now, you should simply include the cast at the top of your `paintComponent` methods.

Now you are ready to draw shapes. The `draw` method of the `Graphics2D` class can draw shapes, such as rectangles, ellipses, line segments, polygons, and arcs. Here we draw a rectangle:

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      . . .
      Rectangle box = new Rectangle(5, 10, 20, 30);
      g2.draw(box);
      . . .
   }
}
```

When positioning the shapes, you need to pay attention to the coordinate system. It is different from the one used in mathematics. The origin (0, 0) is at the upper-left corner of the component, and the *y*-coordinate grows downward.



Following is the source code for the `RectangleComponent` class. Note that the `paintComponent` method of the `RectangleComponent` class draws two rectangles. As you can see from the `import` statements, the `Graphics` and `Graphics2D` classes are part of the `java.awt` package.

section_9_2/RectangleComponent.java

```
1  import java.awt.Graphics;
2  import java.awt.Graphics2D;
3  import java.awt.Rectangle;
4  import javax.swing.JComponent;
5
6  /**
7     A component that draws two rectangles.
8  */
9  public class RectangleComponent extends JComponent
10 {
11    public void paintComponent(Graphics g)
12    {
13       // Recover Graphics2D
14       Graphics2D g2 = (Graphics2D) g;
15
```

```
16         // Construct a rectangle and draw it
17         Rectangle box = new Rectangle(5, 10, 20, 30);
18         g2.draw(box);
19
20         // Move rectangle 15 units to the right and 25 units down
21         box.translate(15, 25);
22
23         // Draw moved rectangle
24         g2.draw(box);
25      }
26   }
```

2.9.3 Displaying a Component in a Frame

In a graphical application, you need a frame to show the application, and you need a component for the drawing. In this section, you will see how to combine the two. Follow these steps:

1. Construct a frame object and configure it.
2. Construct an object of your component class:

   ```
   RectangleComponent component = new RectangleComponent();
   ```

3. Add the component to the frame:

   ```
   frame.add(component);
   ```

4. Make the frame visible.

The following listing shows the complete process.

section_9_3/RectangleViewer.java

```
1  import javax.swing.JFrame;
2
3  public class RectangleViewer
4  {
5     public static void main(String[] args)
6     {
7        JFrame frame = new JFrame();
8
9        frame.setSize(300, 400);
10       frame.setTitle("Two rectangles");
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12
13       RectangleComponent component = new RectangleComponent();
14       frame.add(component);
15
16       frame.setVisible(true);
17    }
18 }
```

Note that the rectangle drawing program consists of two classes:

- The `RectangleComponent` class, whose `paintComponent` method produces the drawing.
- The `RectangleViewer` class, whose `main` method constructs a frame and a `RectangleComponent`, adds the component to the frame, and makes the frame visible.

39. How do you display a square frame with a title bar that reads "Hello, World!"?
40. How can a program display two frames at once?
41. How do you modify the program to draw two squares?
42. How do you modify the program to draw one rectangle and one square?
43. What happens if you call `g.draw(box)` instead of `g2.draw(box)`?

Practice It Now you can try these exercises at the end of the chapter: R2.20, R2.24, E2.17.

2.10 Ellipses, Lines, Text, and Color

In Section 2.9 you learned how to write a program that draws rectangles. In the following sections, you will learn how to draw other shapes: ellipses and lines. With these graphical elements, you can draw quite a few interesting pictures.

You can make simple drawings out of lines, rectangles, and circles.

2.10.1 Ellipses and Circles

To draw an ellipse, you specify its bounding box (see Figure 22) in the same way that you would specify a rectangle, namely by the x- and y-coordinates of the top-left corner and the width and height of the box.

The `Ellipse2D.Double` and `Line2D.Double` classes describe graphical shapes.

However, there is no simple `Ellipse` class that you can use. Instead, you must use one of the two classes `Ellipse2D.Float` and `Ellipse2D.Double`, depending on whether you want to store the ellipse coordinates as single- or double-precision floating-point values. Because the latter are more convenient to use in Java, we will always use the `Ellipse2D.Double` class.

Here is how you construct an ellipse:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(x, y, width, height);
```

The class name `Ellipse2D.Double` looks different from the class names that you have encountered up to now. It consists of two class names `Ellipse2D` and `Double` separated

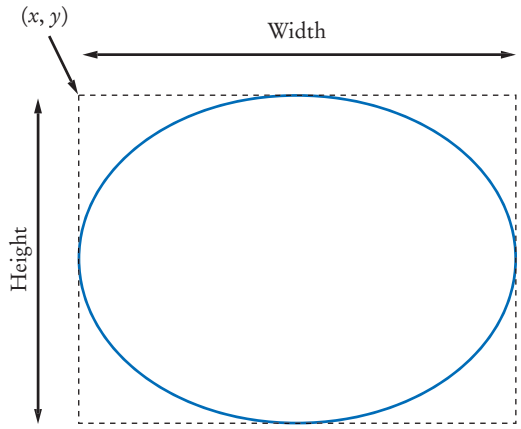


Figure 22 An Ellipse and Its Bounding Box

by a period (.). This indicates that `Ellipse2D.Double` is a so-called **inner class** inside `Ellipse2D`. When constructing and using ellipses, you don't actually need to worry about the fact that `Ellipse2D.Double` is an inner class—just think of it as a class with a long name. However, in the `import` statement at the top of your program, you must be careful that you import only the outer class:

```
import java.awt.geom.Ellipse2D;
```

Drawing an ellipse is easy: Use exactly the same `draw` method of the `Graphics2D` class that you used for drawing rectangles.

```
g2.draw(ellipse);
```

To draw a circle, simply set the width and height to the same values:

```
Ellipse2D.Double circle = new Ellipse2D.Double(x, y, diameter, diameter);
g2.draw(circle);
```

Notice that (*x*, *y*) is the top-left corner of the bounding box, not the center of the circle.

2.10.2 Lines

To draw a line, use an object of the `Line2D.Double` class. A line is constructed by specifying its two end points. You can do this in two ways. Give the *x*- and *y*-coordinates of both end points:

```
Line2D.Double segment = new Line2D.Double(x1, y1, x2, y2);
```

Or specify each end point as an object of the `Point2D.Double` class:

```
Point2D.Double from = new Point2D.Double(x1, y1);
Point2D.Double to = new Point2D.Double(x2, y2);

Line2D.Double segment = new Line2D.Double(from, to);
```

The second option is more object-oriented and is often more useful, particularly if the point objects can be reused elsewhere in the same drawing.

2.10.3 Drawing Text

The `drawString` method draws a string, starting at its basepoint.

You often want to put text inside a drawing, for example, to label some of the parts. Use the `drawString` method of the `Graphics2D` class to draw a string anywhere in a window. You must specify the string and the *x*- and *y*-coordinates of the basepoint of the first character in the string (see Figure 23). For example,

```
g2.drawString("Message", 50, 100);
```

Figure 23 Basepoint and Baseline

2.10.4 Colors

When you first start drawing, all shapes and strings are drawn with a black pen. To change the color, you need to supply an object of type `Color`. Java uses the RGB color model. That is, you specify a color by the amounts of the primary colors—red, green, and blue—that make up the color. The amounts are given as integers between 0 (primary color not present) and 255 (maximum amount present). For example,

```
Color magenta = new Color(255, 0, 255);
```

constructs a `Color` object with maximum red, no green, and maximum blue, yielding a bright purple color called magenta.

For your convenience, a variety of colors have been declared in the `Color` class. Table 4 shows those colors and their RGB values. For example, `Color.PINK` has been declared to be the same color as `new Color(255, 175, 175)`.

When you set a new color in the graphics context, it is used for subsequent drawing operations.

To draw a shape in a different color, first set the color of the `Graphics2D` object, then call the `draw` method:

```
g2.setColor(Color.RED);
g2.draw(circle); // Draws the shape in red
```

If you want to color the inside of the shape, use the `fill` method instead of the `draw` method. For example,

```
g2.fill(circle);
```

fills the inside of the circle with the current color.

Table 4 Predefined Colors

Color	RGB Values
`Color.BLACK`	`0, 0, 0`
`Color.BLUE`	`0, 0, 255`
`Color.CYAN`	`0, 255, 255`
`Color.GRAY`	`128, 128, 128`
`Color.DARK_GRAY`	`64, 64, 64`
`Color.LIGHT_GRAY`	`192, 192, 192`
`Color.GREEN`	`0, 255, 0`
`Color.MAGENTA`	`255, 0, 255`
`Color.ORANGE`	`255, 200, 0`
`Color.PINK`	`255, 175, 175`
`Color.RED`	`255, 0, 0`
`Color.WHITE`	`255, 255, 255`
`Color.YELLOW`	`255, 255, 0`

Figure 24 An Alien Face

The following program puts all these shapes to work, creating a simple drawing (see Figure 24).

section_10/FaceComponent.java

```
1  import java.awt.Color;
2  import java.awt.Graphics;
3  import java.awt.Graphics2D;
4  import java.awt.Rectangle;
5  import java.awt.geom.Ellipse2D;
6  import java.awt.geom.Line2D;
7  import javax.swing.JComponent;
8
9  /**
10    A component that draws an alien face.
11 */
12 public class FaceComponent extends JComponent
13 {
14    public void paintComponent(Graphics g)
15    {
16       // Recover Graphics2D
17       Graphics2D g2 = (Graphics2D) g;
18
19       // Draw the head
20       Ellipse2D.Double head = new Ellipse2D.Double(5, 10, 100, 150);
21       g2.draw(head);
22
23       // Draw the eyes
24       g2.setColor(Color.GREEN);
25       Rectangle eye = new Rectangle(25, 70, 15, 15);
26       g2.fill(eye);
27       eye.translate(50, 0);
28       g2.fill(eye);
29
30       // Draw the mouth
31       Line2D.Double mouth = new Line2D.Double(30, 110, 80, 110);
32       g2.setColor(Color.RED);
33       g2.draw(mouth);
34
35       // Draw the greeting
36       g2.setColor(Color.BLUE);
37       g2.drawString("Hello, World!", 5, 175);
38    }
39 }
```

section_10/FaceViewer.java

```
1  import javax.swing.JFrame;
2
3  public class FaceViewer
4  {
5     public static void main(String[] args)
6     {
7        JFrame frame = new JFrame();
8        frame.setSize(150, 250);
9        frame.setTitle("An Alien Face");
10       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11
12       FaceComponent component = new FaceComponent();
13       frame.add(component);
14
15       frame.setVisible(true);
16    }
17 }
```

44. Give instructions to draw a circle with center (100, 100) and radius 25.

45. Give instructions to draw a letter "V" by drawing two line segments.

46. Give instructions to draw a string consisting of the letter "V".

47. What are the RGB color values of `Color.BLUE`?

48. How do you draw a yellow square on a red background?

Practice It Now you can try these exercises at the end of the chapter: R2.25, E2.18, E2.19.

CHAPTER SUMMARY

Identify objects, methods, and classes.

- Objects are entities in your program that you manipulate by calling methods.
- A method is a sequence of instructions that accesses the data of an object.
- A class describes a set of objects with the same behavior.

Write variable declarations and assignments.

- A variable is a storage location with a name.
- When declaring a variable, you usually specify an initial value.
- When declaring a variable, you also specify the type of its values.
- Use the `int` type for numbers that cannot have a fractional part.
- Use the `double` type for floating-point numbers.
- Numbers can be combined by arithmetic operators such as `+`, `-`, and `*`.
- By convention, variable names should start with a lowercase letter.
- Use comments to add explanations for humans who read your code. The compiler ignores comments.

- Use the assignment operator (=) to change the value of a variable.
- All variables must be initialized before you access them.
- The assignment operator = does *not* denote mathematical equality.

Recognize arguments and return values of methods.

- The public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.
- An argument is a value that is supplied in a method call.
- The return value of a method is a result that the method has computed.

Use constructors to construct new objects.

- Use the `new` operator, followed by a class name and arguments, to construct new objects.

Classify methods as accessor and mutator methods.

- An accessor method does not change the internal data of the object on which it is invoked. A mutator method changes the data.

Use the API documentation for finding method descriptions and packages.

- The API (Application Programming Interface) documentation lists the classes and methods of the Java library.
- Java classes are grouped into packages. Use the `import` statement to use classes that are declared in other packages.

Write programs that test the behavior of methods.

- A test program verifies that methods behave as expected.
- Determining the expected result in advance is an important part of testing.

Describe how multiple object references can refer to the same object.

- An object reference describes the location of an object.
- Multiple object variables can contain references to the same object.
- Number variables store numbers. Object variables store references.

Write programs that display frame windows.

- To show a frame, construct a `JFrame` object, set its size, and make it visible.
- In order to display a drawing in a frame, declare a class that extends the `JComponent` class.

- Place drawing instructions inside the `paintComponent` method. That method is called whenever the component needs to be repainted.
- Use a cast to recover the `Graphics2D` object from the `Graphics` argument of the `paintComponent` method.

Use the Java API for drawing simple figures.

- The `Ellipse2D.Double` and `Line2D.Double` classes describe graphical shapes.
- The `drawString` method draws a string, starting at its basepoint.
- When you set a new color in the graphics context, it is used for subsequent drawing operations.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.awt.Color
java.awt.Component
   getHeight
   getWidth
   setSize
   setVisible
java.awt.Frame
   setTitle
java.awt.geom.Ellipse2D.Double
java.awt.geom.Line2D.Double
java.awt.geom.Point2D.Double
java.awt.Graphics
   setColor
java.awt.Graphics2D
   draw
   drawString
   fill
java.awt.Rectangle
   getX
   getY
   getHeight
   getWidth
   setSize
   translate
java.lang.String
   length
   replace
   toLowerCase
   toUpperCase
javax.swing.JComponent
   paintComponent
javax.swing.JFrame
   setDefaultCloseOperation
```

REVIEW QUESTIONS

- **R2.1** Explain the difference between an object and a class.
- **R2.2** What is the *public interface* of a class? How does it differ from the *implementation* of a class?
- **R2.3** Declare and initialize variables for holding the price and the description of an article that is available for sale.
- **R2.4** What is the value of `mystery` after this sequence of statements?

```
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

- **R2.5** What is wrong with the following sequence of statements?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

■■ **R2.6** Explain the difference between the = symbol in Java and in mathematics.

■■ **R2.7** Give an example of a method that has an argument of type `int`. Give an example of a method that has a return value of type `int`. Repeat for the type `String`.

■■ **R2.8** Write Java statements that initialize a string `message` with "`Hello`" and then change it to "`HELLO`". Use the `toUpperCase` method.

■■ **R2.9** Write Java statements that initialize a string `message` with "`Hello`" and then change it to "`hello`". Use the `replace` method.

■ **R2.10** Explain the difference between an object and an object variable.

■■ **R2.11** Give the Java code for constructing an *object* of class `Rectangle`, and for declaring an *object variable* of class `Rectangle`.

■■ **R2.12** Give Java code for objects with the following descriptions:

a. A rectangle with center (100, 100) and all side lengths equal to 50

b. A string with the contents "Hello, Dave"

Create objects, not object variables.

■■ **R2.13** Repeat Exercise R2.12, but now declare object variables that are initialized with the required objects.

■■ **R2.14** Write a Java statement to initialize a variable `square` with a rectangle object whose top left corner is (10, 20) and whose sides all have length 40. Then write a statement that replaces `square` with a rectangle of the same size and top left corner (20, 20).

■■ **R2.15** Write Java statements that initialize two variables `square1` and `square2` to refer to the same square with center (20, 20) and side length 40.

■■ **R2.16** Find the errors in the following statements:

a.
```
Rectangle r = (5, 10, 15, 20);
```

b.
```
double width = Rectangle(5, 10, 15, 20).getWidth();
```

c.
```
Rectangle r;
r.translate(15, 25);
```

d.
```
r = new Rectangle();
r.translate("far, far away!");
```

■ **R2.17** Name two accessor methods and two mutator methods of the `Rectangle` class.

■■ **R2.18** Consult the API documentation to find methods for

- Concatenating two strings, that is, making a string consisting of the first string, followed by the second string.
- Removing leading and trailing white space of a string.
- Converting a rectangle to a string.
- Computing the smallest rectangle that contains two given rectangles.
- Returning a random floating-point number.

For each method, list the class in which it is defined, the return type, the method name, and the types of the arguments.

■ **R2.19** Explain the difference between an object and an object reference.

■ **Graphics R2.20** What is the difference between a console application and a graphical application?

■■ **Graphics R2.21** Who calls the `paintComponent` method of a component? When does the call to the `paintComponent` method occur?

■■ **Graphics R2.22** Why does the argument of the `paintComponent` method have type `Graphics` and not `Graphics2D`?

■■ **Graphics R2.23** What is the purpose of a graphics context?

■■ **Graphics R2.24** Why are separate viewer and component classes used for graphical programs?

■ **Graphics R2.25** How do you specify a text color?

PRACTICE EXERCISES

■ **Testing E2.1** Write an `AreaTester` program that constructs a `Rectangle` object and then computes and prints its area. Use the `getWidth` and `getHeight` methods. Also print the expected answer.

■ **Testing E2.2** Write a `PerimeterTester` program that constructs a `Rectangle` object and then computes and prints its perimeter. Use the `getWidth` and `getHeight` methods. Also print the expected answer.

E2.3 Write a program that constructs a rectangle with area 42 and a rectangle with perimeter 42. Print the widths and heights of both rectangles.

■■ **Testing E2.4** Look into the API documentation of the `Rectangle` class and locate the method

```
void add(int newx, int newy)
```

Read through the method documentation. Then determine the result of the following statements:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
box.add(0, 0);
```

Write a program `AddTester` that prints the expected and actual location, width, and height of `box` after the call to `add`.

■■ **Testing E2.5** Write a program `ReplaceTester` that encodes a string by replacing all letters `"i"` with `"!"` and all letters `"s"` with `"$"`. Use the `replace` method. Demonstrate that you can correctly encode the string `"Mississippi"`. Print both the actual and expected result.

■■■ **E2.6** Write a program `HollePrinter` that switches the letters `"e"` and `"o"` in a string. Use the `replace` method repeatedly. Demonstrate that the string `"Hello, World!"` turns into `"Holle, Werld!"`

■ **Testing E2.7** The `StringBuilder` class has a method for reversing a string. In a `ReverseTester` class, construct a `StringBuilder` from a given string (such as `"desserts"`), call the `reverse` method followed by the `toString` method, and print the result. Also print the expected value.

■■ **E2.8** In the Java library, a color is specified by its red, green, and blue components between 0 and 255 (see Table 4 on page 68). Write a program `BrighterDemo` that constructs a `Color` object with red, green, and blue values of 50, 100, and 150. Then apply the `brighter` method of the `Color` class and print the red, green, and blue values of the resulting color. (You won't actually see the color—see Exercise E2.9 on how to display the color.)

■■ **Graphics E2.9** Repeat Exercise E2.8, but place your code into the following class. Then the color will be displayed.

```
import java.awt.Color;
import javax.swing.JFrame;

public class BrighterDemo
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      frame.setSize(200, 200);
      Color myColor = ...;
      frame.getContentPane().setBackground(myColor);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

■■ **E2.10** Repeat Exercise E2.8, but apply the `darker` method of the `Color` class twice to the object `Color.RED`. Call your class `DarkerDemo`.

■■ **E2.11** The `Random` class implements a *random number generator*, which produces sequences of numbers that appear to be random. To generate random integers, you construct an object of the `Random` class, and then apply the `nextInt` method. For example, the call `generator.nextInt(6)` gives you a random number between 0 and 5.

Write a program `DieSimulator` that uses the `Random` class to simulate the cast of a die, printing a random number between 1 and 6 every time that the program is run.

■■ **E2.12** Write a program `RandomPrice` that prints a random price between $10.00 and $19.95 every time the program is run.

■■ **Testing E2.13** Look at the API of the `Point` class and find out how to construct a `Point` object. In a `PointTester` program, construct two points with coordinates (3, 4) and (–3, –4). Find the distance between them, using the `distance` method. Print the distance, as well as the expected value. (Draw a sketch on graph paper to find the value you will expect.)

■ **E2.14** Using the `Day` class of Worked Example 2.1, write a `DayTester` program that constructs a `Day` object representing today, adds ten days to it, and then computes the difference between that day and today. Print the difference and the expected value.

■■ **E2.15** Using the `Picture` class of Worked Example 2.2, write a `HalfSizePicture` program that loads a picture and shows it at half the original size, centered in the window.

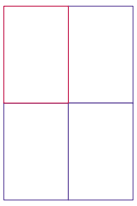
■■ **E2.16** Using the `Picture` class of Worked Example 2.2, write a `DoubleSizePicture` program that loads a picture, doubles its size, and shows the center of the picture in the window.

■■ **Graphics E2.17** Write a graphics program that draws two squares, both with the same center. Provide a class `TwoSquareViewer` and a class `TwoSquareComponent`.

■■ **Graphics E2.18** Write a program that draws two solid squares: one in pink and one in purple. Use a standard color for one of them and a custom color for the other. Provide a class `TwoSquareViewer` and a class `TwoSquareComponent`.

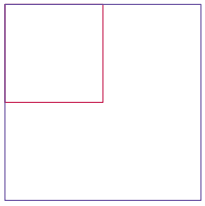
■■ **Graphics E2.19** Write a graphics program that draws your name in red, contained inside a blue rectangle. Provide a class `NameViewer` and a class `NameComponent`.

PROGRAMMING PROJECTS



■■ **P2.1** Write a program called `FourRectanglePrinter` that constructs a `Rectangle` object, prints its location by calling `System.out.println(box)`, and then translates and prints it three more times, so that, if the rectangles were drawn, they would form one large rectangle, as shown at right.

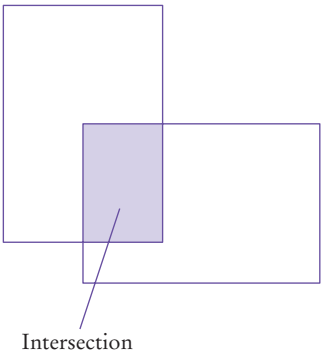
Your program will not produce a drawing. It will simply print the locations of the four rectangles.



■■ **P2.2** Write a `GrowSquarePrinter` program that constructs a `Rectangle` object `square` representing a square with top-left corner (100, 100) and side length 50, prints its location by calling `System.out.println(square)`, applies the `translate` and `grow` methods, and calls `System.out.println(square)` again. The calls to `translate` and `grow` should modify the square so that it has twice the size and the same top-left corner as the original. If the squares were drawn, they would look like the figure at right.

Your program will not produce a drawing. It will simply print the locations of `square` before and after calling the mutator methods.

Look up the description of the `grow` method in the API documentation.



■■■ **P2.3** The `intersection` method computes the *intersection* of two rectangles—that is, the rectangle that would be formed by two overlapping rectangles if they were drawn, as shown at right.

You call this method as follows:

```
Rectangle r3 = r1.intersection(r2);
```

Write a program `IntersectionPrinter` that constructs two rectangle objects, prints them as described in Exercise P2.1, and then prints the rectangle object that describes the intersection. Then the program should print the result of the `intersection` method when the rectangles do not overlap. Add a comment to your program that explains how you can tell whether the resulting rectangle is empty.

■■■ **Graphics P2.4** In this exercise, you will explore a simple way of visualizing a `Rectangle` object. The `setBounds` method of the `JFrame` class moves a frame window to a given rectangle. Complete the following program to visually show the `translate` method of the `Rectangle` class:

```
import java.awt.Rectangle;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class TranslateDemo
{
   public static void main(String[] args)
   {
      // Construct a frame and show it
      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
         frame.setVisible(true);

         // Your work goes here:
         // Construct a rectangle and set the frame bounds

         JOptionPane.showMessageDialog(frame, "Click OK to continue");

         // Your work goes here:
         // Move the rectangle and set the frame bounds again
      }
   }
```

■■■ **P2.5** Write a program `LotteryPrinter` that picks a combination in a lottery. In this lottery, players can choose 6 numbers (possibly repeated) between 1 and 49. Construct an object of the `Random` class and invoke an appropriate method to generate each number. (In a real lottery, repetitions aren't allowed, but we haven't yet discussed the programming constructs that would be required to deal with that problem.) Your program should print out a sentence such as "Play this combination—it'll make you rich!", followed by a lottery combination.

■■ **P2.6** Using the `Day` class of Worked Example 1, write a program that generates a `Day` object representing February 28 of this year, and three more such objects that represent February 28 of the next three years. Advance each object by one day, and print each object. Also print the expected values:

```
2012-02-29
Expected: 2012-02-29
2013-03-01
Expected: 2013-03-01
. . .
```

■■■ **P2.7** The `GregorianCalendar` class describes a point in time, as measured by the Gregorian calendar, the standard calendar that is commonly used throughout the world today. You construct a `GregorianCalendar` object from a year, month, and day of the month, like this:

```
GregorianCalendar cal = new GregorianCalendar(); // Today's date
GregorianCalendar eckertsBirthday = new GregorianCalendar(1919,
      Calendar.APRIL, 9);
```

Use the values `Calendar.JANUARY . . . Calendar.DECEMBER` to specify the month.

The `add` method can be used to add a number of days to a `GregorianCalendar` object:

```
cal.add(Calendar.DAY_OF_MONTH, 10); // Now cal is ten days from today
```

This is a mutator method—it changes the `cal` object.

The `get` method can be used to query a given `GregorianCalendar` object:

```
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
int month = cal.get(Calendar.MONTH);
int year = cal.get(Calendar.YEAR);
int weekday = cal.get(Calendar.DAY_OF_WEEK);
   // 1 is Sunday, 2 is Monday, . . . , 7 is Saturday
```

Your task is to write a program that prints:

- The date and weekday that is 100 days from today.
- The weekday of your birthday.
- The date that is 10,000 days from your birthday.

Use the birthday of a computer scientist if you don't want to reveal your own birthday.

Hint: The `GregorianCalendar` class is complex, and it is a really good idea to write a few test programs to explore the API before tackling the whole problem. Start with a program that constructs today's date, adds ten days, and prints out the day of the month and the weekday.

■■■ **Testing P2.8** Write a program `LineDistanceTester` that constructs a line joining the points (100, 100) and (200, 200), then constructs points (100, 200), (150, 150), and (250, 50). Print the distance from the line to each of the three points, using the `ptSegDist` method of the `Line2D` class. Also print the expected values. (Draw a sketch on graph paper to find what values you expect.)

■■ **Graphics P2.9** Repeat Exercise P2.8, but now write a graphical application that shows the line and the points. Draw each point as a tiny circle. Use the `drawString` method to draw each distance next to the point, using calls

```
g2.drawString("Distance: " + distance, p.getX(), p.getY());
```

■■ **Graphics P2.10** Write a graphics program that draws 12 strings, one each for the 12 standard colors (except `Color.WHITE`), each in its own color. Provide a class `ColorNameViewer` and a class `ColorNameComponent`.

■■ **Graphics P2.11** Write a program to plot the face at right. Provide a class `FaceViewer` and a class `FaceComponent`.

■■ **Graphics P2.12** Write a graphical program that draws a traffic light.

■■ **Graphics P2.13** Run the following program:

```
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class FrameViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      frame.setSize(200, 200);
      JLabel label = new JLabel("Hello, World!");
      label.setOpaque(true);
      label.setBackground(Color.PINK);
      frame.add(label);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Modify the program as follows:

- Double the frame size.
- Change the greeting to "Hello, *your name*!".
- Change the background color to pale green (see Exercise E2.9).
- For extra credit, add an image of yourself. (*Hint:* Construct an `ImageIcon`.)

ANSWERS TO SELF-CHECK QUESTIONS

1. Objects with the same behavior belong to the same class. A window lets in light while protecting a room from the outside wind and heat or cold. A water heater has completely different behavior. It heats water. They belong to different classes.
2. When one calls a method, one is not concerned with how it does its job. As long as a light bulb illuminates a room, it doesn't matter to the occupant how the photons are produced.
3. There are three errors:
   - You cannot have spaces in variable names.
   - The variable type should be `double` because it holds a fractional value.
   - There is a semicolon missing at the end of the statement.
4.
```
double unitPrice = 1.95;
int quantity = 2;
```
5.
```
System.out.print("Total price: ");
System.out.println(unitPrice * quantity);
```
6. `int` and `String`
7. `double`
8. Only the first two are legal identifiers.
9. `String myName = "John Q. Public";`
10. No, the left-hand side of the `=` operator must be a variable.
11. `greeting = "Hello, Nina!";`

    Note that

    `String greeting = "Hello, Nina!";`

    is not the right answer—that statement declares a new variable.
12. Assignment would occur when one car is replaced by another in the parking space.
13. `river.length()` or `"Mississippi".length()`
14. `System.out.println(greeting.toUpperCase());`

    or

    ```
    System.out.println(
       "Hello, World!".toUpperCase());
    ```
15. It is not legal. The variable `river` has type `String`. The `println` method is not a method of the `String` class.
16. The arguments are the strings "p" and "s".
17. `"Missississi"`
18. 12
19. As `public String toUpperCase()`, with no argument and return type `String`.
20. `new Rectangle(90, 90, 20, 20)`
21.
```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = new Rectangle(25, 10, 20, 30);
```
22. 0
23. `new PrintStream("output.txt");`
24. `PrintStream out = new PrintStream("output.txt");`
25.
```
Before: 5
After: 30
```
26.
```
Before: 20
After: 20
```

    Moving the rectangle does not affect its width or height. You can change the width and height with the `setSize` method.
27.
```
HELLO
hello
```

    Note that calling `toUpperCase` doesn't modify the string.
28. An accessor—it doesn't modify the original string but returns a new string with uppercase letters.
29. `box.translate(-5, -10)`, provided the method is called immediately after storing the new rectangle into `box`.

30. `toLowerCase`

31. `"Hello, Space !"`—only the leading and trailing spaces are trimmed.

32. The arguments of the `translate` method tell how far to move the rectangle in the *x*- and *y*-directions. The arguments of the `setLocation` method indicate the new *x*- and *y*-values for the top-left corner.

For example, `box.move(1, 1)` moves the `box` one pixel down and to the right. `box.setLocation(1, 1)` moves `box` to the top-left corner of the screen.

33. Add the statement `import java.util.Random;` at the top of your program.

34. In the `java.math` package.

35. `x: 30, y: 25`

36. Because the `translate` method doesn't modify the shape of the rectangle.

37. Now `greeting` and `greeting2` both refer to the same `String` object.

38. Both variables still refer to the same string, and the string has not been modified. Recall that the `toUpperCase` method constructs a new string that contains uppercase characters, leaving the original string unchanged.

39. Modify the `EmptyFrameViewer` program as follows:

```
frame.setSize(300, 300);
frame.setTitle("Hello, World!");
```

40. Construct two `JFrame` objects, set each of their sizes, and call `setVisible(true)` on each of them.

41. Change line 17 of `RectangleComponent` to

```
Rectangle box = new Rectangle(5, 10, 20, 20);
```

42. Replace the call to `box.translate(15, 25)` with

```
box = new Rectangle(20, 35, 20, 20);
```

43. The compiler complains that `g` doesn't have a `draw` method.

44. `g2.draw(new Ellipse2D.Double(75, 75, 50, 50));`

45.
```
Line2D.Double segment1
   = new Line2D.Double(0, 0, 10, 30);
g2.draw(segment1);
Line2D.Double segment2
   = new Line2D.Double(10, 30, 20, 0);
g2.draw(segment2);
```

46. `g2.drawString("V", 0, 30);`

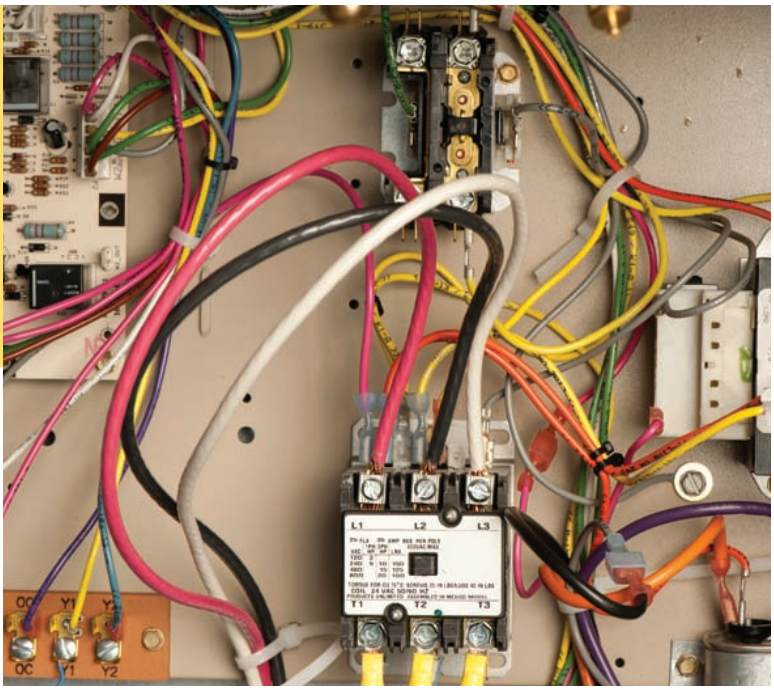
47. `0, 0, 255`

48. First fill a big red square, then fill a small yellow square inside:

```
g2.setColor(Color.RED);
g2.fill(new Rectangle(0, 0, 200, 200));
g2.setColor(Color.YELLOW);
g2.fill(new Rectangle(50, 50, 100, 100));
```

CHAPTER 3

IMPLEMENTING CLASSES



CHAPTER GOALS

To become familiar with the process of implementing classes

To be able to implement and test simple methods

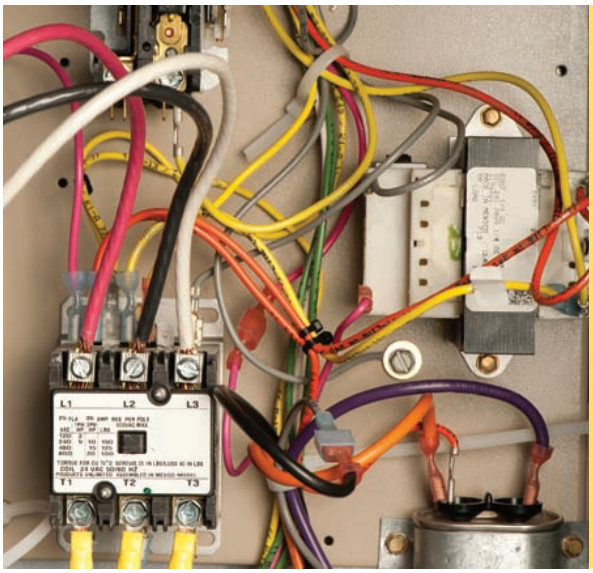
To understand the purpose and use of constructors

To understand how to access instance variables and local variables

To be able to write javadoc comments

To implement classes for drawing graphical shapes

CHAPTER CONTENTS



In this chapter, you will learn how to implement your own classes. You will start with a given design that specifies the public interface of the class—that is, the methods through which programmers can manipulate the objects of the class. Then you will learn the steps to completing the class—creating the internal "workings" like the inside of an air conditioner shown here. You need to implement the methods, which entails finding a data representation for the objects and supplying the instructions for each method. You need to document your efforts so that other programmers can understand and use your creation. And you need to provide a tester to validate that your class works correctly.

3.1 Instance Variables and Encapsulation

In Chapter 2, you learned how to use objects from existing classes. In this chapter, you will start implementing your own classes. We begin with a very simple example that shows you how objects store their data, and how methods access the data of an object. Our first example is a class that models a *tally counter*, a mechanical device that is used to count people—for example, to find out how many people attend a concert or board a bus (see Figure 1).

Figure 1 A Tally Counter

3.1.1 Instance Variables

Whenever the operator clicks the button of a tally counter, the counter value advances by one. We model this operation with a `click` method of a `Counter` class. A physical counter has a display to show the current value. In our simulation, we use a `getValue` method to get the current value. For example,

```
Counter tally = new Counter();
tally.click();
tally.click();
int result = tally.getValue(); // Sets result to 2
```

When implementing the `Counter` class, you need to determine the data that each counter object contains. In this simple example, that is very straightforward. Each counter needs a variable that keeps track of the number of simulated button clicks.

An object's instance variables store the data required for executing its methods.

An object stores its data in **instance variables**. An *instance* of a class is an object of the class. Thus, an instance variable is a storage location that is present in each object of the class.

You specify instance variables in the class declaration:

```
public class Counter
{
   private int value;
   . . .
}
```

Syntax 3.1 Instance Variable Declaration

Syntax

```
public class ClassName
{
   private typeName variableName;
   . . .
}
```

```
public class Counter
{
   private int value;
   . . .
}
```

Instance variables should always be private.

Each object of this class has a separate copy of this instance variable.

Type of the variable

An instance variable declaration consists of the following parts:

- An **access specifier** (`private`)
- The **type** of the instance variable (such as `int`)
- The name of the instance variable (such as `value`)

Each object of a class has its own set of instance variables.

Each object of a class has its own set of instance variables. For example, if `concertCounter` and `boardingCounter` are two objects of the `Counter` class, then each object has its own `value` variable (see Figure 2). As you will see in Section 3.3, the instance variable `value` is set to 0 when a `Counter` object is constructed.

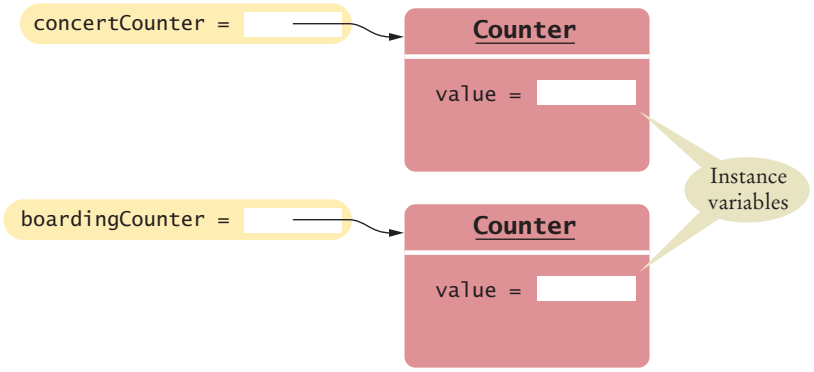


Figure 2
Instance Variables

These clocks have common behavior, but each of them has a different state. Similarly, objects of a class can have their instance variables set to different values.

3.1.2 The Methods of the Counter Class

In this section, we will look at the implementation of the methods of the `Counter` class.

The `click` method advances the counter value by 1. You have seen the method header syntax in Chapter 2. Now, focus on the body of the method inside the braces.

```
public void click()
{
   value = value + 1;
}
```

Note how the `click` method accesses the instance variable `value`. *Which* instance variable? The one belonging to the object on which the method is invoked. For example, consider the call

```
concertCounter.click();
```

This call advances the `value` variable of the `concertCounter` object.

The `getValue` method returns the current value:

```
public int getValue()
{
   return value;
}
```

The `return` statement is a special statement that terminates the method call and returns a result (the **return value**) to the method's caller.

Private instance variables can only be accessed by methods of the same class.

Instance variables are generally declared with the access specifier `private`. That specifier means that they can be accessed only by the methods of the *same class*, not by any other method. For example, the `value` variable can be accessed by the `click` and `getValue` methods of the `Counter` class but not by a method of another class. Those other methods need to use the `Counter` class methods if they want to manipulate a counter's internal data.

3.1.3 Encapsulation

In the preceding section, you learned that you should hide instance variables by making them private. Why would a programmer want to hide something?

The strategy of information hiding is not unique to computer programming—it is used in many engineering disciplines. Consider the thermostat that you find in your home. It is a device that allows a user to set temperature preferences and that controls the furnace and the air conditioner. If you ask your contractor what is inside the thermostat, you will likely get a shrug.

The thermostat is a *black box*, something that magically does its thing. A contractor would never open the control module—it contains electronic parts that can only be serviced at the factory. In general, engineers use the term "black box" to describe any device whose inner workings are hidden. Note that a black box is not totally mysterious. Its interface with the outside world is well-defined. For example, the contractor understands how the thermostat must be connected with the furnace and air conditioner.

Encapsulation is the process of hiding implementation details and providing methods for data access.

The process of hiding implementation details while publishing an interface is called **encapsulation**. In Java, the `class` construct provides encapsulation. The public methods of a class are the interface through which the private implementation is manipulated.

Why do contractors use prefabricated components such as thermostats and furnaces? These "black boxes" greatly simplify the work of the contractor. In ancient times, builders had to know how to construct furnaces from brick and mortar, and how to produce some rudimentary temperature controls. Nowadays, a contractor just makes a trip to the hardware store, without needing to know what goes on inside the components.

A thermostat functions as a "black box" whose inner workings are hidden.

Similarly, a programmer using a class is not burdened by unnecessary detail, as you know from your own experience. In Chapter 2, you used classes for strings, streams, and windows without worrying how these classes are implemented.

Encapsulation allows a programmer to use a class without having to know its implementation.

Encapsulation also helps with diagnosing errors. A large program may consist of hundreds of classes and thousands of methods, but if there is an error with the internal data of an object, you only need to look at the methods of one class. Finally, encapsulation makes it possible to change the implementation of a class without having to tell the programmers who use the class.

Information hiding makes it simpler for the implementor of a class to locate errors and change implementations.

In Chapter 2, you learned to be an object user. You saw how to obtain objects, how to manipulate them, and how to assemble them into a program. In that chapter, you treated objects as black boxes. Your role was roughly analogous to the contractor who installs a new thermostat.

In this chapter, you will move on to implementing classes. In these sections, your role is analogous to the hardware manufacturer who puts together a thermostat from buttons, sensors, and other electronic parts. You will learn the necessary Java programming techniques that enable your objects to carry out the desired behavior.

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a demonstration of the `Counter` class.

section_1/Counter.java

```
1  /**
2     This class models a tally counter.
3  */
4  public class Counter
5  {
6     private int value;
7
8     /**
9        Gets the current value of this counter.
10       @return the current value
11    */
12    public int getValue()
13    {
14       return value;
15    }
16
17    /**
18       Advances the value of this counter by 1.
19    */
20    public void click()
21    {
22       value = value + 1;
23    }
24
```

```
25    /**
26       Resets the value of this counter to 0.
27    */
28    public void reset()
29    {
30       value = 0;
31    }
32 }
```

1. Supply the body of a method `public void unclick()` that undoes an unwanted button click.
2. Suppose you use a class `Clock` with private instance variables `hours` and `minutes`. How can you access these variables in your program?
3. Consider the `Counter` class. A counter's value starts at `0` and is advanced by the `click` method, so it should never be negative. Suppose you found a negative `value` variable during testing. Where would you look for the error?
4. In Chapters 1 and 2, you used `System.out` as a black box to cause output to appear on the screen. Who designed and implemented `System.out`?
5. Suppose you are working in a company that produces personal finance software. You are asked to design and implement a class for representing bank accounts. Who will be the users of your class?

Practice It Now you can try these exercises at the end of the chapter: R3.1, R3.3, E3.1.

3.2 Specifying the Public Interface of a Class

In the following sections, we will discuss the process of specifying the public interface of a class. Imagine that you are a member of a team that works on banking software. A fundamental concept in banking is a *bank account*. Your task is to design a `BankAccount` class that can be used by other programmers to manipulate bank accounts. What methods should you provide? What information should you give the programmers who use this class? You will want to settle these questions before you implement the class.

3.2.1 Specifying Methods

In order to implement a class, you first need to know which methods are required.

You need to know exactly what operations of a bank account need to be implemented. Some operations are essential (such as taking deposits), whereas others are not important (such as giving a gift to a customer who opens a bank account). Deciding which operations are essential is not always an easy task. We will revisit that issue in Chapters 8 and 12. For now, we will assume that a competent designer has decided that the following are considered the essential operations of a bank account:

- Deposit money
- Withdraw money
- Get the current balance

In Java, you call a method when you want to apply an operation to an object. To figure out the exact specification of the method calls, imagine how a programmer would carry out the bank account operations. We'll assume that the variable `harrysChecking` contains a reference to an object of type `BankAccount`. We want to support method calls such as the following:

```
harrysChecking.deposit(2240.59);
harrysChecking.withdraw(500);
double currentBalance = harrysChecking.getBalance();
```

The first two methods are mutators. They modify the balance of the bank account and don't return a value. The third method is an accessor. It returns a value that you store in a variable or pass to a method.

From the sample calls, we decide the `BankAccount` class should declare three methods:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance()`

Recall from Chapter 2 that `double` denotes the double-precision floating-point type, and `void` indicates that a method does not return a value.

Here we only give the method *headers*. When you declare a method, you also need to provide the method **body**, which consists of statements that are executed when the method is called.

```
public void deposit(double amount)
{
   method body—implementation filled in later
}
```

We will supply the method bodies in Section 3.3.

Note that the methods have been declared as `public`, indicating that all other methods in a program can call them. Occasionally, it can be useful to have `private` methods. They can only be called from other methods of the same class.

Some people like to fill in the bodies so that they compile, like this:

```
public double getBalance()
{
   // TODO: fill in implementation
   return 0;
}
```

That is a good idea if you compose your specification in your development environment—you won't get warnings about incorrect code.

3.2.2 Specifying Constructors

Constructors set the initial data for objects.

As you know from Chapter 2, constructors are used to initialize objects. In Java, a **constructor** is very similar to a method, with two important differences:

- The name of the constructor is always the same as the name of the class (e.g., `BankAccount`).
- Constructors have no return type (not even `void`).

We want to be able to construct bank accounts that initially have a zero balance, as well as accounts that have a given initial balance.

For this purpose, we specify two constructors:

- `public BankAccount()`
- `public BankAccount(double initialBalance)`

They are used as follows:

```
BankAccount harrysChecking = new BankAccount();
BankAccount momsSavings = new BankAccount(5000);
```

The constructor name is always the same as the class name.

Don't worry about the fact that there are two constructors with the same name—*all* constructors of a class have the same name, that is, the name of the class. The compiler can tell them apart because they take different arguments. The first constructor takes no arguments at all. Such a constructor is called a **no-argument constructor**. The second constructor takes an argument of type `double`.

Just like a method, a constructor also has a body—a sequence of statements that is executed when a new object is constructed.

```
public BankAccount()
{
   constructor body—implementation filled in later
}
```

The statements in the constructor body will set the instance variables of the object that is being constructed—see Section 3.3.

When declaring a class, you place all constructor and method declarations inside, like this:

```
public class BankAccount
{
   private instance variables—filled in later

   // Constructors
   public BankAccount()
   {
      implementation—filled in later
   }

   public BankAccount(double initialBalance)
   {
      implementation—filled in later
   }

   // Methods
   public void deposit(double amount)
   {
      implementation—filled in later
   }

   public void withdraw(double amount)
   {
      implementation—filled in later
   }

   public double getBalance()
   {
      implementation—filled in later
   }
}
```

Syntax 3.2 Class Declaration

Syntax

```
accessSpecifier class ClassName
{
   instance variables
   constructors
   methods
}
```

```
public class Counter
{
   private int value;

   public Counter(int initialValue) { value = initialValue; }

   public void click() { value = value + 1; }
   public int getValue() { return value; }
}
```

Public interface

Private implementation

The public constructors and methods of a class form the **public interface** of the class. These are the operations that any programmer can use to create and manipulate `BankAccount` objects.

3.2.3 Using the Public Interface

Our `BankAccount` class is simple, but it allows programmers to carry out all of the important operations that commonly occur with bank accounts. For example, consider this program segment, authored by a programmer who uses the `BankAccount` class. These statements transfer an amount of money from one bank account to another:

```
// Transfer from one account to another
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

And here is a program segment that adds interest to a savings account:

```
double interestRate = 5; // 5 percent interest
double interestAmount = momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

As you can see, programmers can use objects of the `BankAccount` class to carry out meaningful tasks, without knowing how the `BankAccount` objects store their data or how the `BankAccount` methods do their work.

Of course, as implementors of the `BankAccount` class, we will need to supply the private implementation. We will do so in Section 3.3. First, however, an important step remains: *documenting* the public interface. That is the topic of the next section.

3.2.4 Commenting the Public Interface

When you implement classes and methods, you should get into the habit of thoroughly *commenting* their behaviors. In Java there is a very useful standard form for

documentation comments. If you use this form in your classes, a program called **javadoc** can automatically generate a neat set of HTML pages that describe them. (See Programming Tip 3.1 on page 92 for a description of this utility.)

Use documentation comments to describe the classes and public methods of your programs.

A documentation comment is placed before the class or method declaration that is being documented. It starts with a `/**`, a special comment delimiter used by the `javadoc` utility. Then you describe the method's *purpose*. Then, for each argument, you supply a line that starts with `@param`, followed by the name of the variable that holds the argument (which is called a **parameter variable**). Supply a short explanation for each argument after the variable name. Finally, you supply a line that starts with `@return`, describing the return value. You omit the `@param` tag for methods that have no arguments, and you omit the `@return` tag for methods whose return type is `void`.

The `javadoc` utility copies the *first* sentence of each comment to a summary table in the HTML documentation. Therefore, it is best to write that first sentence with some care. It should start with an uppercase letter and end with a period. It does not have to be a grammatically complete sentence, but it should be meaningful when it is pulled out of the comment and displayed in a summary.

Here are two typical examples:

```
/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
   implementation—filled in later
}

/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   implementation—filled in later
}
```

The comments you have just seen explain individual *methods*. Supply a brief comment for each *class*, too, explaining its purpose. Place the documentation comment above the class declaration:

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   . . .
}
```

Your first reaction may well be "Whoa! Am I supposed to write all this stuff?" Sometimes, documentation comments seem pretty repetitive, but in most cases, they are informative. Even with seemingly repetitive comments, you should take the time to write them.

It is always a good idea to write the method comment *first*, before writing the code in the method body. This is an excellent test to see that you firmly understand what

you need to program. If you can't explain what a class or method does, you aren't ready to implement it.

Provide documentation comments for every class, every method, every parameter variable, and every return value.

What about very simple methods? You can easily spend more time pondering whether a comment is too trivial to write than it takes to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without any comment can cause real grief to future maintenance programmers. According to the standard Java documentation style, *every* class, *every* method, *every* parameter variable, and *every* return value should have a comment.

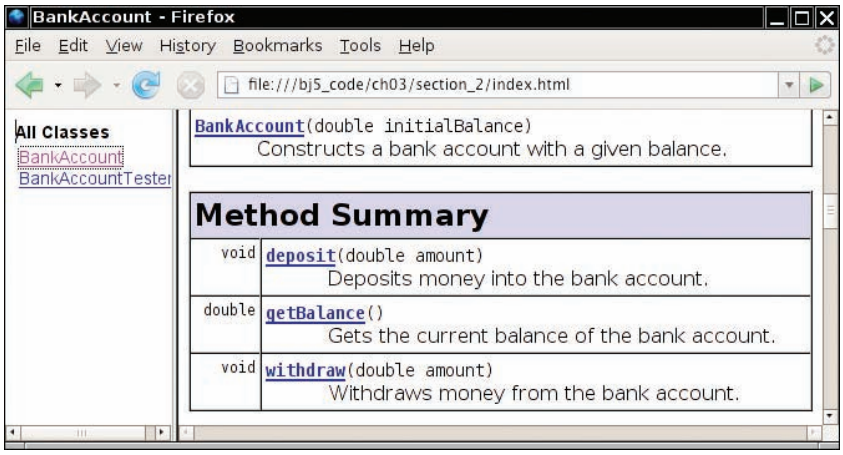


Figure 3 A Method Summary Generated by `javadoc`

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download the `BankAccount` class with documentation but without implementation.

The `javadoc` utility formats your comments into a neat set of documents that you can view in a web browser. It makes good use of the seemingly repetitive phrases. The first sentence of the comment is used for a *summary table* of all methods of your class (see Figure 3). The `@param` and `@return` comments are neatly formatted in the detail description of each method (see Figure 4). If you omit any of the comments, then `javadoc` generates documents that look strangely empty.

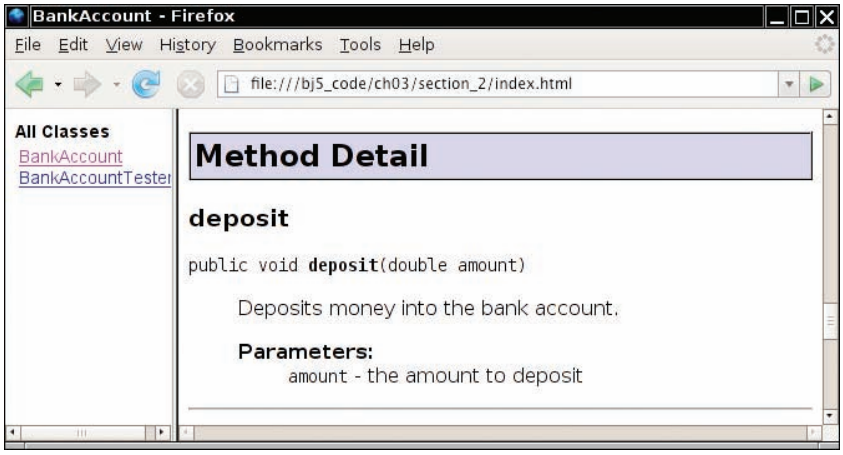


Figure 4 Method Detail Generated by `javadoc`

This documentation format should look familiar. The programmers who implement the Java library use `javadoc` themselves. They too document every class, every method, every parameter variable, and every return value, and then use `javadoc` to extract the documentation in HTML format.

6. How can you use the methods of the public interface to *empty* the `harrysChecking` bank account?
7. What is wrong with this sequence of statements?

```
BankAccount harrysChecking = new BankAccount(10000);
System.out.println(harrysChecking.withdraw(500));
```

8. Suppose you want a more powerful bank account abstraction that keeps track of an *account number* in addition to the balance. How would you change the public interface to accommodate this enhancement?
9. Suppose we enhance the `BankAccount` class so that each account has an account number. Supply a documentation comment for the constructor

```
public BankAccount(int accountNumber, double initialBalance)
```

10. Why is the following documentation comment questionable?

```
/**
   Each account has an account number.
   @return the account number of this account
*/
public int getAccountNumber()
```

Practice It Now you can try these exercises at the end of the chapter: R3.7, R3.8, R3.9.

Common Error 3.1

Declaring a Constructor as void

Do not use the `void` reserved word when you declare a constructor:

```
public void BankAccount()  // Error—don't use void!
```

This would declare a method with return type `void` and *not* a constructor. Unfortunately, the Java compiler does not consider this a syntax error.

Programming Tip 3.1

The javadoc Utility

Always insert documentation comments in your code, whether or not you use `javadoc` to produce HTML documentation. Most people find the HTML documentation convenient, so it is worth learning how to run `javadoc`. Some programming environments (such as BlueJ) can execute `javadoc` for you. Alternatively, you can invoke the `javadoc` utility from a shell window, by issuing the command

```
javadoc MyClass.java
```

or, if you want to document multiple Java files,

```
javadoc *.java
```

The `javadoc` utility produces files such as `MyClass.html` in HTML format, which you can inspect in a browser. If you know HTML (see Appendix H), you can embed HTML tags into the

comments to specify fonts or add images. Perhaps most importantly, `javadoc` automatically provides *hyperlinks* to other classes and methods.

You can run `javadoc` before implementing any methods. Just leave all the method bodies empty. Don't run the compiler—it would complain about missing return values. Simply run `javadoc` on your file to generate the documentation for the public interface that you are about to implement.

The `javadoc` tool is wonderful because it does one thing right: It allows you to put the documentation *together with your code*. That way, when you update your programs, you can see right away which documentation needs to be updated. Hopefully, you will update it right then and there. Afterward, run `javadoc` again and get updated information that is timely and nicely formatted.

3.3 Providing the Class Implementation

Now that you understand the specification of the public interface of the `BankAccount` class, let's provide the implementation.

3.3.1 Providing Instance Variables

The private implementation of a class consists of instance variables, and the bodies of constructors and methods.

First, we need to determine the data that each bank account object contains. In the case of our simple bank account class, each object needs to store a single value, the current balance. (A more complex bank account class might store additional data—perhaps an account number, the interest rate paid, the date for mailing out the next statement, and so on.)

```
public class BankAccount
{
   private double balance;
   // Methods and constructors below
   . . .
}
```

In general, it can be challenging to find a good set of instance variables. Ask yourself what an object needs to remember so that it can carry out any of its methods.

Like a wilderness explorer who needs to carry all items that may be needed, an object needs to store the data required for its method calls.

3.3.2 Providing Constructors

A **constructor** has a simple job: to initialize the instance variables of an object.

Recall that we designed the `BankAccount` class to have two constructors. The first constructor simply sets the balance to zero:

```
public BankAccount()
{
   balance = 0;
}
```

The second constructor sets the balance to the value supplied as the construction argument:

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

To see how these constructors work, let us trace the statement

```
BankAccount harrysChecking = new BankAccount(1000);
```

one step at a time.

Here are the steps that are carried out when the statement executes (see Figure 5):

- Create a new object of type `BankAccount`. 1
- Call the second constructor (because an argument is supplied in the constructor call).
- Set the parameter variable `initialBalance` to 1000. 2
- Set the `balance` instance variable of the newly created object to `initialBalance`. 3
- Return an object reference, that is, the memory location of the object, as the value of the `new` expression.
- Store that object reference in the `harrysChecking` variable. 4

In general, when you implement constructors, be sure that each constructor initializes all instance variables, and that you make use of all parameter variables (see Common Error 3.2 on page 98).

A constructor is like a set of assembly instructions for an object.

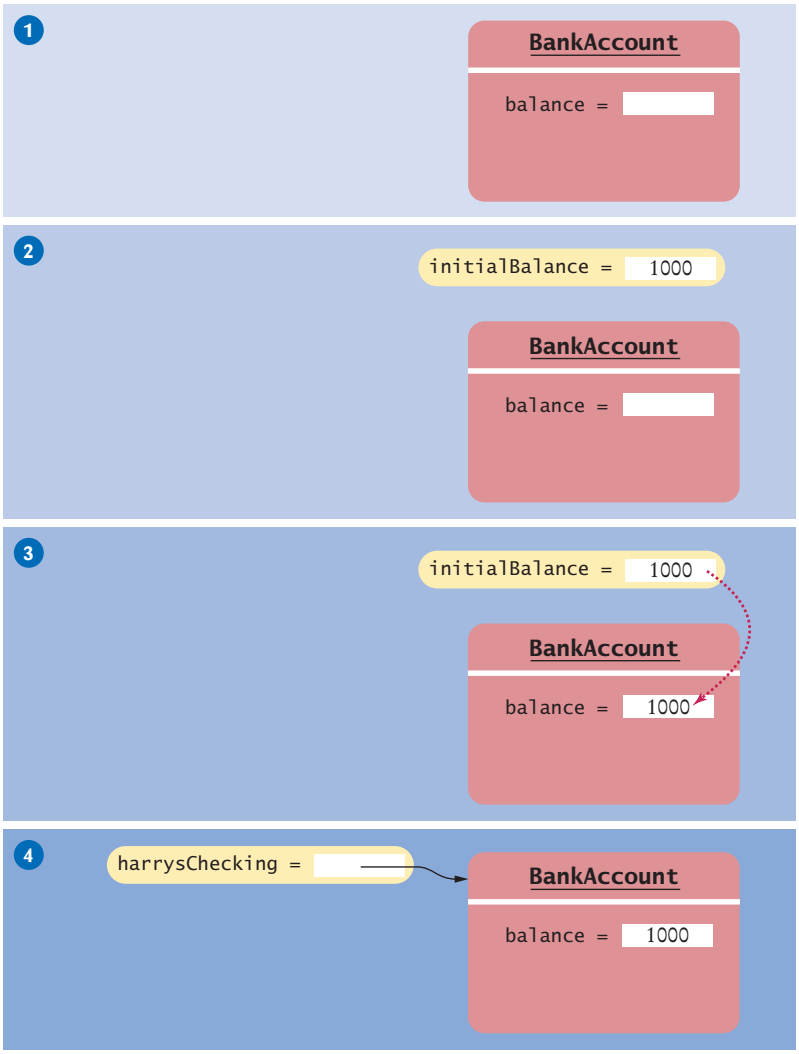


Figure 5
How a Constructor Works

3.3.3 Providing Methods

In this section, we finish implementing the methods of the `BankAccount` class.

When you implement a method, ask yourself whether it is an accessor or mutator method. A mutator method needs to update the instance variables in some way. An accessor method retrieves or computes a result.

Here is the `deposit` method. It is a mutator method, updating the balance:

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

The `withdraw` method is very similar to the `deposit` method:

```
public void withdraw(double amount)
{
   balance = balance - amount;
}
```

Table 1 Implementing Classes

Example	Comments
`public class BankAccount { . . . }`	This is the start of a class declaration. Instance variables, methods, and constructors are placed inside the braces.
`private double balance;`	This is an instance variable of type `double`. Instance variables should be declared as `private`.
`public double getBalance() { . . . }`	This is a method declaration. The body of the method must be placed inside the braces.
`. . . { return balance; }`	This is the body of the `getBalance` method. The `return` statement returns a value to the caller of the method.
`public void deposit(double amount) { . . . }`	This is a method with a parameter variable (`amount`). Because the method is declared as `void`, it has no return value.
`. . . { balance = balance + amount; }`	This is the body of the `deposit` method. It does not have a `return` statement.
`public BankAccount() { . . . }`	This is a constructor declaration. A constructor has the same name as the class and no return type.
`. . . { balance = 0; }`	This is the body of the constructor. A constructor should initialize the instance variables.

There is one method left, `getBalance`. Unlike the `deposit` and `withdraw` methods, which modify the instance variable of the object on which they are invoked, the `getBalance` method returns a value:

```
public double getBalance()
{
   return balance;
}
```

We have now completed the implementation of the `BankAccount` class—see the code listing below. There is only one step remaining: testing that the class works correctly. That is the topic of the next section.

section_3/BankAccount.java

```
1  /**
2     A bank account has a balance that can be changed by
3     deposits and withdrawals.
4  */
5  public class BankAccount
6  {
7     private double balance;
8
9     /**
10       Constructs a bank account with a zero balance.
11    */
12    public BankAccount()
13    {
14       balance = 0;
15    }
```

```
16
17       /**
18          Constructs a bank account with a given balance.
19          @param initialBalance the initial balance
20       */
21       public BankAccount(double initialBalance)
22       {
23          balance = initialBalance;
24       }
25
26       /**
27          Deposits money into the bank account.
28          @param amount the amount to deposit
29       */
30       public void deposit(double amount)
31       {
32          balance = balance + amount;
33       }
34
35       /**
36          Withdraws money from the bank account.
37          @param amount the amount to withdraw
38       */
39       public void withdraw(double amount)
40       {
41          balance = balance - amount;
42       }
43
44       /**
45          Gets the current balance of the bank account.
46          @return the current balance
47       */
48       public double getBalance()
49       {
50          return balance;
51       }
52    }
```

11. Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance variables?

12. Why does the following code not succeed in robbing mom's bank account?

```
public class BankRobber
{
   public static void main(String[] args)
   {
      BankAccount momsSavings = new BankAccount(1000);
      momsSavings.balance = 0;
   }
}
```

13. The `Rectangle` class has four instance variables: `x`, `y`, `width`, and `height`. Give a possible implementation of the `getWidth` method.

14. Give a possible implementation of the `translate` method of the `Rectangle` class.

Practice It Now you can try these exercises at the end of the chapter: R3.4, R3.10, E3.4.

Common Error 3.2

Ignoring Parameter Variables

A surprisingly common beginner's error is to ignore parameter variables of methods or constructors. This usually happens when an assignment gives an example with specific values. For example, suppose you are asked to provide a class `Letter` with a recipient and a sender, and you are given a sample letter like this:

```
Dear John:

I am sorry we must part.
I wish you all the best.

Sincerely,

Mary
```

Now look at this incorrect attempt:

```
public class Letter
{
   private String recipient;
   private String sender;

   public Letter(String aRecipient, String aSender)
   {
      recipient = "John"; // Error—should use parameter variable
      sender = "Mary"; // Same error
   }
   . . .
}
```

The constructor ignores the names of the recipient and sender arguments that were provided to the constructor. If a user constructs a

```
new Letter("John", "Yoko")
```

the sender is still set to `"Mary"`, which is bound to be embarrassing.

The constructor should use the parameter variables, like this:

```
public Letter(String aRecipient, String aSender)
{
   recipient = aRecipient;
   sender = aSender;
}
```

HOW TO 3.1

Implementing a Class

This "How To" section tells you how you implement a class from a given specification.

Problem Statement Implement a class that models a self-service cash register. The customer scans the price tags and deposits money in the machine. The machine dispenses the change.

Step 1 Find out which methods you are asked to supply.

In a simulation, you won't have to provide every feature that occurs in the real world—there are too many. In the cash register example, we don't deal with sales tax or credit card payments. The assignment tells you *which aspects* of the self-service cash register your class should simulate. Make a list of them:

- Process the price of each purchased item.
- Receive payment.
- Calculate the amount of change due to the customer.

Step 2 Specify the public interface.

Turn the list in Step 1 into a set of methods, with specific types for the parameter variables and the return values. Many programmers find this step simpler if they write out method calls that are applied to a sample object, like this:

```
CashRegister register = new CashRegister();
register.recordPurchase(29.95);
register.recordPurchase(9.95);
register.receivePayment(50);
double change = register.giveChange();
```

Now we have a specific list of methods:

- `public void recordPurchase(double amount)`
- `public void receivePayment(double amount)`
- `public double giveChange()`

To complete the public interface, you need to specify the constructors. Ask yourself what information you need in order to construct an object of your class. Sometimes you will want two constructors: one that sets all instance variables to a default and one that sets them to user-supplied values.

In the case of the cash register example, we can get by with a single constructor that creates an empty register. A more realistic cash register might start out with some coins and bills so that we can give exact change, but that is well beyond the scope of our assignment.

Thus, we add a single constructor:

- `public CashRegister()`

Step 3 Document the public interface.

Here is the documentation, with comments, that describes the class and its methods:

```
/**
   A cash register totals up sales and computes change due.
*/
public class CashRegister
{
   /**
      Constructs a cash register with no money in it.
   */
   public CashRegister()
   {
   }

   /**
      Records the sale of an item.
      @param amount the price of the item
   */
   public void recordPurchase(double amount)
   {
```

```
   }

   /**
      Processes a payment received from the customer.
      @param amount the amount of the payment
   */
   public void receivePayment(double amount)
   {
   }

   /**
      Computes the change due and resets the machine for the next customer.
      @return the change due to the customer
   */
   public double giveChange()
   {
   }
}
```

Step 4 Determine instance variables.

Ask yourself what information an object needs to store to do its job. Remember, the methods can be called in any order. The object needs to have enough internal memory to be able to process every method using just its instance variables and the parameter variables. Go through each method, perhaps starting with a simple one or an interesting one, and ask yourself what you need to carry out the method's task. Make instance variables to store the information that the method needs.

Just as importantly, don't introduce unnecessary instance variables (see Common Error 3.3). If a value can be computed from other instance variables, it is generally better to compute it on demand than to store it.

In the cash register example, you need to keep track of the total purchase amount and the payment. You can compute the change due from these two amounts.

```
public class CashRegister
{
   private double purchase;
   private double payment;
   . . .
}
```

Step 5 Implement constructors and methods.

Implement the constructors and methods in your class, one at a time, starting with the easiest ones. Here is the implementation of the `recordPurchase` method:

```
public void recordPurchase(double amount)
{
   purchase = purchase + amount;
}
```

The `receivePayment` method looks almost the same,

```
public void receivePayment(double amount)
{
   payment = payment + amount;
}
```

but why does the method add the amount, instead of simply setting `payment = amount`? A customer might provide two separate payments, such as two $10 bills, and the machine must process them both. Remember, methods can be called more than once, and they can be called in any order.

Finally, here is the `giveChange` method. This method is a bit more sophisticated—it computes the change due, and it also resets the cash register for the next sale.

```
public double giveChange()
{
   double change = payment - purchase;
   purchase = 0;
   payment = 0;
   return change;
}
```

If you find that you have trouble with the implementation, you may need to rethink your choice of instance variables. It is common for a beginner to start out with a set of instance variables that cannot accurately reflect the state of an object. Don't hesitate to go back and add or modify instance variables.

You can find the complete implementation in the `how_to_1` directory of the book's companion code.

Step 6 Test your class.

Write a short tester program and execute it. The tester program should carry out the method calls that you found in Step 2.

```
public class CashRegisterTester
{
   public static void main(String[] args)
   {
      CashRegister register = new CashRegister();

      register.recordPurchase(29.50);
      register.recordPurchase(9.25);
      register.receivePayment(50);

      double change = register.giveChange();

      System.out.println(change);
      System.out.println("Expected: 11.25");
   }
}
```

The output of this test program is:

```
11.25
Expected: 11.25
```

WORKED EXAMPLE 3.1 **Making a Simple Menu**

Learn how to implement a class that constructs simple text-based menus. Go to `wiley.com/go/javaexamples` and download Worked Example 3.1.

3.4 Unit Testing

An engineer tests a part in isolation. This is an example of unit testing.

A unit test verifies that a class works correctly in isolation, outside a complete program.

In the preceding section, we completed the implementation of the `BankAccount` class. What can you do with it? Of course, you can compile the file `BankAccount.java`. However, you can't *execute* the resulting `BankAccount.class` file. It doesn't contain a `main` method. That is normal—most classes don't contain a `main` method.

In the long run, your class may become a part of a larger program that interacts with users, stores data in files, and so on. However, before integrating a class into a program, it is always a good idea to test it in isolation. Testing in isolation, outside a complete program, is called **unit testing**.

To test your class, you have two choices. Some interactive development environments have commands for constructing objects and invoking methods (see Special Topic 2.1). Then you can test a class simply by constructing an object, calling methods, and verifying that you get the expected return values. Figure 6 shows the result of calling the `getBalance` method on a `BankAccount` object in BlueJ.

To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.

Alternatively, you can write a *tester class*. A tester class is a class with a `main` method that contains statements to run methods of another class. As discussed in Section 2.7, a tester class typically carries out the following steps:

1. Construct one or more objects of the class that is being tested.
2. Invoke one or more methods.
3. Print out one or more results.
4. Print the expected results.

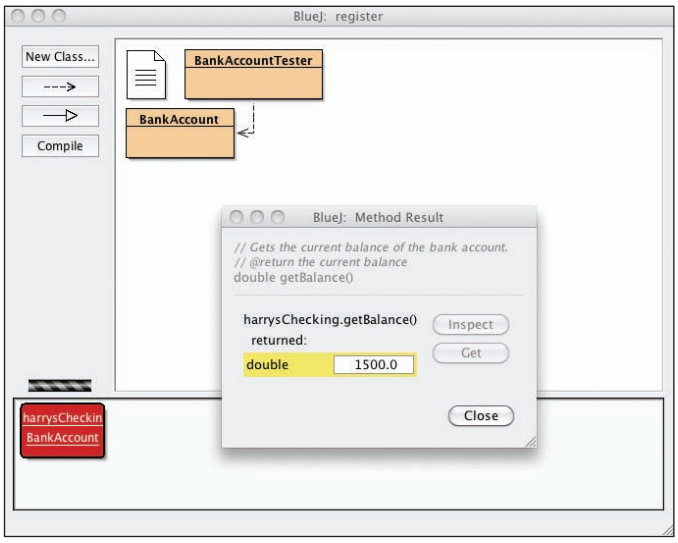


Figure 6 The Return Value of the `getBalance` Method in BlueJ

The `MoveTester` class in Section 2.7 is a good example of a tester class. That class runs methods of the `Rectangle` class—a class in the Java library.

Following is a class to run methods of the `BankAccount` class. The `main` method constructs an object of type `BankAccount`, invokes the `deposit` and `withdraw` methods, and then displays the remaining balance on the console.

We also print the value that we expect to see. In our sample program, we deposit $2,000 and withdraw $500. We therefore expect a balance of $1,500.

section_4/BankAccountTester.java

```
1  /**
2     A class to test the BankAccount class.
3  */
4  public class BankAccountTester
5  {
6     /**
7        Tests the methods of the BankAccount class.
8        @param args not used
9     */
10    public static void main(String[] args)
11    {
12       BankAccount harrysChecking = new BankAccount();
13       harrysChecking.deposit(2000);
14       harrysChecking.withdraw(500);
15       System.out.println(harrysChecking.getBalance());
16       System.out.println("Expected: 1500");
17    }
18 }
```

Program Run

```
1500
Expected: 1500
```

To produce a program, you need to combine the `BankAccount` and the `BankAccountTester` classes. The details for building the program depend on your compiler and development environment. In most environments, you need to carry out these steps:

1. Make a new subfolder for your program.
2. Make two files, one for each class.
3. Compile both files.
4. Run the test program.

Many students are surprised that such a simple program contains two classes. However, this is normal. The two classes have entirely different purposes. The `BankAccount` class describes objects that compute bank balances. The `BankAccountTester` class runs a test that puts a `BankAccount` object through its paces.

15. When you run the `BankAccountTester` program, how many objects of class `BankAccount` are constructed? How many objects of type `BankAccountTester`?
16. Why is the `BankAccountTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

Practice It Now you can try these exercises at the end of the chapter: E3.3, E3.10.

Computing & Society 3.1 Electronic Voting Machines

In the 2000 presidential elections in the United States, votes were tallied by a variety of machines. Some machines processed cardboard ballots into which voters punched holes to indicate their choices (see below). When voters were not careful, remains of paper—the now infamous "chads"—were partially stuck in the punch cards, causing votes to be miscounted. A manual recount was necessary, but it was not carried out everywhere due to time constraints and procedural wrangling. The election was very close, and there remain doubts in the minds of many people whether the election outcome would have been different if the voting machines had accurately counted the intent of the voters.

Punch Card Ballot

Subsequently, voting machine manufacturers have argued that electronic voting machines would avoid the problems caused by punch cards or optically scanned forms. In an electronic voting machine, voters indicate their preferences by pressing buttons or touching icons on a computer screen. Typically, each voter is presented with a summary screen for review before casting the ballot. The process is very similar to using a bank's automated teller machine.

It seems plausible that these machines make it more likely that a vote is counted in the same way that the voter intends. However, there has been significant controversy surrounding some types of electronic voting machines. If a machine simply records the votes and prints out the totals after the election has been completed, then how do you know that the machine worked correctly? Inside the machine is a computer that executes a program, and, as you may know from your own experience, programs can have bugs.

In fact, some electronic voting machines do have bugs. There have been isolated cases where machines reported tallies that were impossible. When a machine reports far more or far fewer votes than voters, then it is clear that it malfunctioned. Unfortunately, it is then impossible to find out the actual votes. Over time, one would expect these bugs to be fixed in the software. More insidiously, if the results are plausible, nobody may ever investigate.

Many computer scientists have spoken out on this issue and confirmed that it is impossible, with today's technology, to tell that software is error free and has not been tampered with. Many of them recommend that electronic voting machines should employ a *voter verifiable audit trail*. (A good source of information is `http://verifiedvoting.org`.) Typically, a voter-verifiable machine prints out a ballot. Each voter has a chance to review the printout, and then deposits it in an old-fashioned ballot box. If there is a problem with the electronic equipment, the printouts can be scanned or counted by hand.

Touch Screen Voting Machine

As this book is written, this concept is strongly resisted both by manufacturers of electronic voting machines and by their customers, the cities and counties that run elections. Manufacturers are reluctant to increase the cost of the machines because they may not be able to pass the cost increase on to their customers, who tend to have tight budgets. Election officials fear problems with malfunctioning printers, and some of them have publicly stated that they actually prefer equipment that eliminates bothersome recounts.

What do you think? You probably use an automated bank teller machine to get cash from your bank account. Do you review the paper record that the machine issues? Do you check your bank statement? Even if you don't, do you put your faith in other people who double-check their balances, so that the bank won't get away with widespread cheating?

Is the integrity of banking equipment more important or less important than that of voting machines? Won't every voting process have some room for error and fraud anyway? Is the added cost for equipment, paper, and staff time reasonable to combat a potentially slight risk of malfunction and fraud? Computer scientists cannot answer these questions—an informed society must make these tradeoffs. But, like all professionals, they have an obligation to speak out and give accurate testimony about the capabilities and limitations of computing equipment.

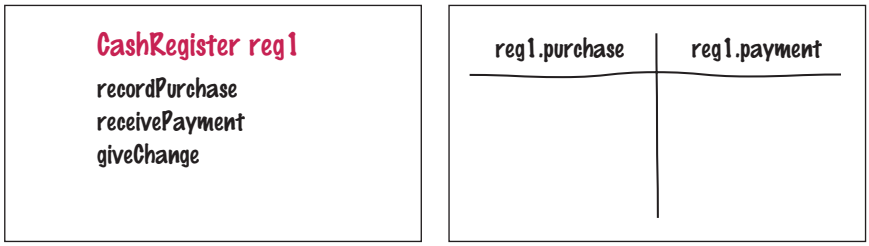
3.5 Problem Solving: Tracing Objects

Researchers have studied why some students have an easier time learning how to program than others. One important skill of successful programmers is the ability to simulate the actions of a program with pencil and paper. In this section, you will see how to develop this skill by tracing method calls on objects.

Write the methods on the front of a card and the instance variables on the back.

Use an index card or a sticky note for each object. On the front, write the methods that the object can execute. On the back, make a table for the values of the instance variables.

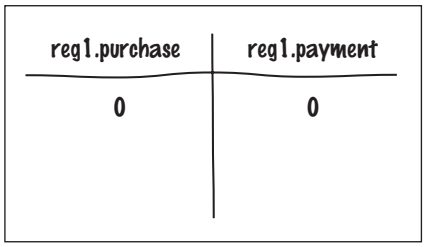
Here is a card for a `CashRegister` object:



front *back*

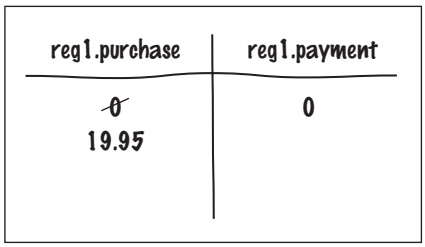
In a small way, this gives you a feel for encapsulation. An object is manipulated through its public interface (on the front of the card), and the instance variables are hidden in the back.

When an object is constructed, fill in the initial values of the instance variables:



Update the values of the instance variables when a mutator method is called.

Whenever a mutator method is executed, cross out the old values and write the new ones below. Here is what happens after a call to the `recordPurchase` method:



If you have more than one object in your program, you will have multiple cards, one for each object:

reg1.purchase	reg1.payment
~~0~~ 19.95	~~0~~ 19.95

reg2.purchase	reg2.payment
~~0~~ ~~29.50~~ 9.25	~~0~~ 50.00

These diagrams are also useful when you design a class. Suppose you are asked to enhance the `CashRegister` class to compute the sales tax. Add methods `recordTaxablePurchase` and `getSalesTax` to the front of the card. Now turn the card over, look over the instance variables, and ask yourself whether the object has sufficient information to compute the answer. Remember that each object is an autonomous unit. Any value that can be used in a computation must be

- An instance variable.
- A method argument.
- A static variable (uncommon; see Section 8.4).

To compute the sales tax, we need to know the tax rate and the total of the taxable items. (Food items are usually not subject to sales tax.) We don't have that information available. Let us introduce additional instance variables for the tax rate and the taxable total. The tax rate can be set in the constructor (assuming it stays fixed for the lifetime of the object). When adding an item, we need to be told whether the item is taxable. If so, we add its price to the taxable total.

For example, consider the following statements.

```
CashRegister reg3(7.5); // 7.5 percent sales tax
reg3.recordPurchase(3.95); // Not taxable
reg3.recordTaxablePurchase(19.95); // Taxable
```

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download an enhanced `CashRegister` class that computes the sales tax.

When you record the effect on a card, it looks like this:

reg3.purchase	reg3.taxablePurchase	reg3.payment	reg3.taxRate
~~0~~ 3.95	~~0~~ 19.95	0	7.5

With this information, we can compute the tax. It is **taxablePurchase x taxRate / 100**. Tracing the object helped us understand the need for additional instance variables.

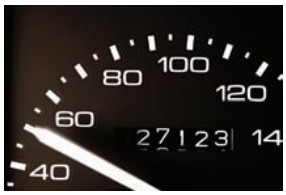
SELF CHECK

17. Consider a `Car` class that simulates fuel consumption in a car. We will assume a fixed efficiency (in miles per gallon) that is supplied in the constructor. There are methods for adding gas, driving a given distance, and checking the amount of gas left in the tank. Make a card for a `Car` object, choosing suitable instance variables and showing their values after the object was constructed.

18. Trace the following method calls:

```
Car myCar(25);
myCar.addGas(20);
myCar.drive(100);
myCar.drive(200);
myCar.addGas(5);
```

19. Suppose you are asked to simulate the odometer of the car, by adding a method `getMilesDriven`. Add an instance variable to the object's card that is suitable for computing this method's result.



20. Trace the methods of Self Check 18, updating the instance variable that you added in Self Check 19.

Practice It Now you can try these exercises at the end of the chapter: R3.18, R3.19, R3.20.

3.6 Local Variables

Local variables are declared in the body of a method.

When a method exits, its local variables are removed.

Instance variables are initialized to a default value, but you must initialize local variables.

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a demonstration of local variables.

In this section, we discuss the behavior of *local* variables. A **local variable** is a variable that is declared in the body of a method. For example, the `giveChange` method in How To 3.1 declares a local variable `change`:

```
public double giveChange()
{
   double change = payment - purchase;
   purchase = 0;
   payment = 0;
   return change;
}
```

Parameter variables are similar to local variables, but they are declared in method headers. For example, the following method declares a parameter variable `amount`:

```
public void receivePayment(double amount)
```

Local and parameter variables belong to methods. When a method runs, its local and parameter variables come to life. When the method exits, they are removed immediately. For example, if you call `register.giveChange()`, then a variable `change` is created. When the method exits, that variable is removed.

In contrast, instance variables belong to objects, not methods. When an object is constructed, its instance variables are created. The instance variables stay alive until no method uses the object any longer. (The Java virtual machine contains an agent called a **garbage collector** that periodically reclaims objects when they are no longer used.)

An important difference between instance variables and local variables is initialization. You must **initialize** all local variables. If you don't initialize a local variable, the compiler complains when you try to use it. (Note that parameter variables are initialized when the method is called.)

Instance variables are initialized with a default value before a constructor is invoked. Instance variables that are numbers are initialized to 0. Object references are set to a special value called `null`. If an object reference is `null`, then it refers to no object at all. We will discuss the `null` value in greater detail in Section 5.2.5.

21. What do local variables and parameter variables have in common? In which essential aspect do they differ?
22. Why was it necessary to introduce the local variable `change` in the `giveChange` method? That is, why didn't the method simply end with the statement

    ```
    return payment - purchase;
    ```
23. Consider a `CashRegister` object `reg1` whose `payment` instance variable has the value 20 and whose `purchase` instance variable has the value 19.5. Trace the call `reg1.giveChange()`. Include the local variable `change`. Draw an X in its column when the variable ceases to exist.

Practice It Now you can try these exercises at the end of the chapter: R3.14, R3.15.

Common Error 3.3

Duplicating Instance Variables in Local Variables

Beginning programmers commonly add types to assignment statements, thereby changing them into local variable declarations. For example,

```
public double giveChange()
{
   double change = payment - purchase;
   double purchase = 0; // ERROR! This declares a local variable.
   double payment = 0;  // ERROR! The instance variable is not updated.
   return change;
}
```

Another common error is to declare a parameter variable with the same name as an instance variable. For example, consider this `BankAccount` constructor:

```
public BankAccount(double balance)
{
   balance = balance; // ERROR! Does not set the instance variable
}
```

This constructor simply sets the parameter variable to itself, leaving it unchanged. A simple remedy is to come up with a different name for the parameter variable:

```
public BankAccount(double initialBalance)
{
   balance = initialBalance; // OK
}
```

Common Error 3.4

Providing Unnecessary Instance Variables

A common beginner's mistake is to use instance variables when local variables would be more appropriate. For example, consider the `change` variable of the `giveChange` method. It is not needed anywhere else—that's why it is local to the method. But what if it had been declared as an instance variable?

```
public class CashRegister
{
   private double purchase;
   private double payment;
   private double change; // Not appropriate
```

```
   public double giveChange()
   {
      change = payment - purchase;
      purchase = 0;
      payment = 0;
      return change;
   }
   ...
}
```

This class will work, but there is a hidden danger. Other methods can read and write to the `change` instance variable, which can be a source of confusion.

Use instance variables for values that an object needs to remember between method calls. Use local variables for values that don't need to be retained when a method has completed.

Common Error 3.5

Forgetting to Initialize Object References in a Constructor

Just as it is a common error to forget to initialize a local variable, it is easy to forget about instance variables. Every constructor needs to ensure that all instance variables are set to appropriate values.

If you do not initialize an instance variable, the Java compiler will initialize it for you. Numbers are initialized with `0`, but object references—such as string variables—are set to the `null` reference.

Of course, `0` is often a convenient default for numbers. However, `null` is hardly ever a convenient default for objects. Consider this "lazy" constructor for a modified version of the `BankAccount` class:

```
public class BankAccount
{
   private double balance;
   private String owner;
   . . .
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
}
```

Then `balance` is initialized, but the `owner` variable is set to a `null` reference. This can be a problem—it is illegal to call methods on the `null` reference.

To avoid this problem, it is a good idea to initialize every instance variable:

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
   owner = "None";
}
```

3.7 The this Reference

When you call a method, you pass two kinds of inputs to the method:

- The object on which you invoke the method
- The method arguments

For example, when you call

```
momsSavings.deposit(500);
```

the `deposit` method needs to know the account object (`momsSavings`) as well as the amount that is being deposited (500).

When you implement the method, you provide a parameter variable for each argument. But you don't need to provide a parameter variable for the object on which the method is being invoked. That object is called the **implicit parameter**. All other parameter variables (such as the amount to be deposited in our example) are called **explicit parameters**.

Look again at the code of the `deposit` method:

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

Here, `amount` is an explicit parameter. You don't see the implicit parameter—that is why it is called "implicit". But consider what `balance` means exactly. After all, our program may have multiple `BankAccount` objects, and *each of them* has its own balance.

Use of an instance variable name in a method denotes the instance variable of the implicit parameter.

Because we are depositing the money into `momsSavings`, `balance` must mean `momsSavings.balance`. In general, when you refer to an instance variable inside a method, it means the instance variable of the implicit parameter.

In any method, you can access the implicit parameter—the object on which the method is called—with the reserved word `this`. For example, in the preceding method invocation, `this` refers to the same object as `momsSavings` (see Figure 7).

The statement

```
balance = balance + amount;
```

The `this` reference denotes the implicit parameter.

actually means

```
this.balance = this.balance + amount;
```

When you refer to an instance variable in a method, the compiler automatically applies it to the `this` reference. Some programmers actually prefer to manually insert the `this` reference before every instance variable because they find it makes the code clearer. Here is an example:

```
public BankAccount(double initialBalance)
{
   this.balance = initialBalance;
}
```

You may want to try it out and see if you like that style.

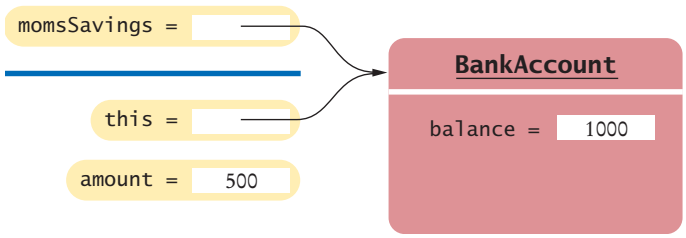


Figure 7 The Implicit Parameter of a Method Call

The this reference can also be used to distinguish between instance variables and local or parameter variables. Consider the constructor

```
public BankAccount(double balance)
{
   this.balance = balance;
}
```

A local variable shadows an instance variable with the same name. You can access the instance variable name through the this reference.

The expression this.balance clearly refers to the balance instance variable. However, the expression balance by itself seems ambiguous. It could denote either the parameter variable or the instance variable. The Java language specifies that in this situation the local variable wins out. It "shadows" the instance variable. Therefore,

```
this.balance = balance;
```

means: "Set the instance variable balance to the parameter variable balance".

There is another situation in which it is important to understand implicit parameters. Consider the following modification to the BankAccount class. We add a method to apply the monthly account fee:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      withdraw(10); // Withdraw $10 from this account
   }
}
```

A method call without an implicit parameter is applied to the same object.

That means to withdraw from the *same* bank account object that is carrying out the monthlyFee operation. In other words, the implicit parameter of the withdraw method is the (invisible) implicit parameter of the monthlyFee method.

If you find it confusing to have an invisible parameter, you can use the this reference to make the method easier to read:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      this.withdraw(10); // Withdraw $10 from this account
   }
}
```

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program that demonstrates the this reference.

You have now seen how to use objects and implement classes, and you have learned some important technical details about variables and method parameters. The remainder of this chapter continues the optional graphics track. In the next chapter, you will learn more about the most fundamental data types of the Java language.

SELF CHECK

24. How many implicit and explicit parameters does the withdraw method of the BankAccount class have, and what are their names and types?

25. In the deposit method, what is the meaning of this.amount? Or, if the expression has no meaning, why not?

26. How many implicit and explicit parameters does the main method of the BankAccountTester class have, and what are they called?

Practice It Now you can try these exercises at the end of the chapter: R3.11, R3.12.

Special Topic 3.1

Calling One Constructor from Another

Consider the `BankAccount` class. It has two constructors: a no-argument constructor to initialize the balance with zero, and another constructor to supply an initial balance. Rather than explicitly setting the balance to zero, one constructor can call another constructor of the same class instead. There is a shorthand notation to achieve this result:

```
public class BankAccount
{
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   public BankAccount()
   {
      this(0);
   }
   . . .
}
```

The command `this(0);` means "Call another constructor of this class and supply the value `0`". Such a call to another constructor can occur only as the *first line in a constructor*.

This syntax is a minor convenience. We will not use it in this book. Actually, the use of the reserved word `this` is a little confusing. Normally, `this` denotes a reference to the implicit parameter, but if `this` is followed by parentheses, it denotes a call to another constructor of the same class.

3.8 Shape Classes

It is a good idea to make a class for any part of a drawing that can occur more than once.

In this section, we continue the optional graphics track by discussing how to organize complex drawings in a more object-oriented fashion.

When you produce a drawing that has multiple shapes, or parts made of multiple shapes, such as the car in Figure 8, it is a good idea to make a separate class for each part. The class should have a `draw` method that draws the shape, and a constructor to set the position of the shape. For example, here is the outline of the `Car` class:

```
public class Car
{
   public Car(int x, int y)
   {
      // Remember position
      . . .
   }

   public void draw(Graphics2D g2)
   {
      // Drawing instructions
      . . .
   }
}
```

You will find the complete class declaration at the end of this section. The `draw` method contains a rather long sequence of instructions for drawing the body, roof, and tires.

Figure 8 The Car Component Draws Two Car Shapes

To figure out how to draw a complex shape, make a sketch on graph paper.

The coordinates of the car parts seem a bit arbitrary. To come up with suitable values, draw the image on graph paper and read off the coordinates (Figure 9).

The program that produces Figure 8 is composed of three classes.

- The `Car` class is responsible for drawing a single car. Two objects of this class are constructed, one for each car.
- The `CarComponent` class displays the drawing.
- The `CarViewer` class shows a frame that contains a `CarComponent`.

Let us look more closely at the `CarComponent` class. The `paintComponent` method draws two cars. We place one car in the top-left corner of the window, and the other car in the bottom-right corner. To compute the bottom-right position, we call the `getWidth` and `getHeight` methods of the `JComponent` class. These methods return the dimensions of the component. We subtract the dimensions of the car to determine the position of `car2`:

```
Car car1 = new Car(0, 0);
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```

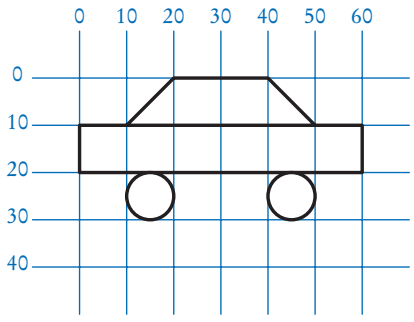


Figure 9
Using Graph Paper to Find Shape Coordinates

Pay close attention to the call to `getWidth` inside the `paintComponent` method of `CarComponent`. The method call has no implicit parameter, which means that the method is applied to the same object that executes the `paintComponent` method. The component simply obtains *its own* width.

Run the program and resize the window. Note that the second car always ends up at the bottom-right corner of the window. Whenever the window is resized, the `paintComponent` method is called and the car position is recomputed, taking the current component dimensions into account.

section_8/CarComponent.java

```
1  import java.awt.Graphics;
2  import java.awt.Graphics2D;
3  import javax.swing.JComponent;
4
5  /**
6     This component draws two car shapes.
7  */
8  public class CarComponent extends JComponent
9  {
10    public void paintComponent(Graphics g)
11    {
12       Graphics2D g2 = (Graphics2D) g;
13
14       Car car1 = new Car(0, 0);
15
16       int x = getWidth() - 60;
17       int y = getHeight() - 30;
18
19       Car car2 = new Car(x, y);
20
21       car1.draw(g2);
22       car2.draw(g2);
23    }
24 }
```

section_8/Car.java

```
1  import java.awt.Graphics2D;
2  import java.awt.Rectangle;
3  import java.awt.geom.Ellipse2D;
4  import java.awt.geom.Line2D;
5  import java.awt.geom.Point2D;
6
7  /**
8     A car shape that can be positioned anywhere on the screen.
9  */
10 public class Car
11 {
12    private int xLeft;
13    private int yTop;
14
15    /**
16       Constructs a car with a given top left corner.
17       @param x the x-coordinate of the top-left corner
18       @param y the y-coordinate of the top-left corner
19    */
20    public Car(int x, int y)
21    {
```

```
22          xLeft = x;
23          yTop = y;
24       }
25
26       /**
27          Draws the car.
28          @param g2 the graphics context
29       */
30       public void draw(Graphics2D g2)
31       {
32          Rectangle body = new Rectangle(xLeft, yTop + 10, 60, 10);
33          Ellipse2D.Double frontTire
34                = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
35          Ellipse2D.Double rearTire
36                = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
37
38          // The bottom of the front windshield
39          Point2D.Double r1 = new Point2D.Double(xLeft + 10, yTop + 10);
40          // The front of the roof
41          Point2D.Double r2 = new Point2D.Double(xLeft + 20, yTop);
42          // The rear of the roof
43          Point2D.Double r3 = new Point2D.Double(xLeft + 40, yTop);
44          // The bottom of the rear windshield
45          Point2D.Double r4 = new Point2D.Double(xLeft + 50, yTop + 10);
46
47          Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
48          Line2D.Double roofTop = new Line2D.Double(r2, r3);
49          Line2D.Double rearWindshield = new Line2D.Double(r3, r4);
50
51          g2.draw(body);
52          g2.draw(frontTire);
53          g2.draw(rearTire);
54          g2.draw(frontWindshield);
55          g2.draw(roofTop);
56          g2.draw(rearWindshield);
57       }
58    }
```

section_8/CarViewer.java

```
1  import javax.swing.JFrame;
2
3  public class CarViewer
4  {
5     public static void main(String[] args)
6     {
7        JFrame frame = new JFrame();
8
9        frame.setSize(300, 400);
10       frame.setTitle("Two cars");
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12
13       CarComponent component = new CarComponent();
14       frame.add(component);
15
16       frame.setVisible(true);
17    }
18 }
```

27. Which class needs to be modified to have the two cars positioned next to each other?
28. Which class needs to be modified to have the car tires painted in black, and what modification do you need to make?
29. How do you make the cars twice as big?

Practice It Now you can try these exercises at the end of the chapter: E3.16, E3.21.

HOW TO 3.2 Drawing Graphical Shapes

Suppose you want to write a program that displays graphical shapes such as cars, aliens, charts, or any other images that can be obtained from rectangles, lines, and ellipses. These instructions give you a step-by-step procedure for decomposing a drawing into parts and implementing a program that produces the drawing.

Problem Statement Create a program that draws a national flag.

Step 1 Determine the shapes that you need for the drawing.

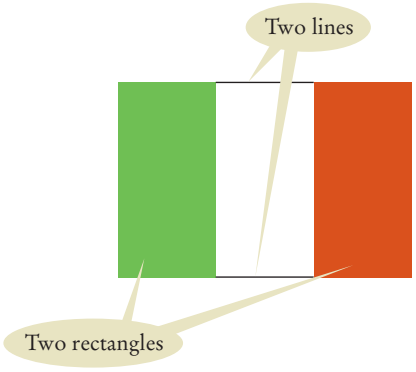
You can use the following shapes:

- Squares and rectangles
- Circles and ellipses
- Lines

The outlines of these shapes can be drawn in any color, and you can fill the insides of these shapes with any color. You can also use text to label parts of your drawing.

Some national flags consist of three equally wide sections of different colors, side by side.

You could draw such a flag using three rectangles. But if the middle rectangle is white, as it is, for example, in the flag of Italy (green, white, red), it is easier and looks better to draw a line on the top and bottom of the middle portion:



Step 2 Find the coordinates for the shapes.

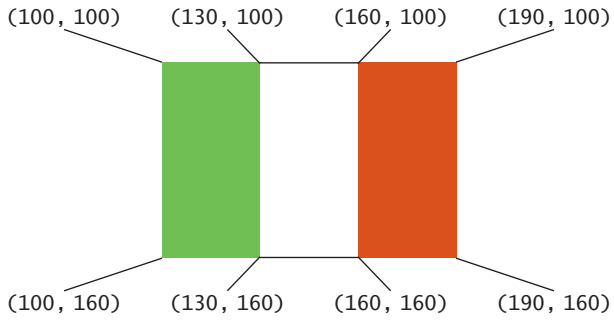
You now need to find the exact positions for the geometric shapes.

- For rectangles, you need the *x*- and *y*-position of the top-left corner, the width, and the height.
- For ellipses, you need the top-left corner, width, and height of the bounding rectangle.
- For lines, you need the *x*- and *y*-positions of the start and end points.
- For text, you need the *x*- and *y*-position of the basepoint.

A commonly-used size for a window is 300 by 300 pixels. You may not want the flag crammed all the way to the top, so perhaps the upper-left corner of the flag should be at point (100, 100).

Many flags, such as the flag of Italy, have a width : height ratio of 3 : 2. (You can often find exact proportions for a particular flag by doing a bit of Internet research on one of several Flags of the World sites.) For example, if you make the flag 90 pixels wide, then it should be 60 pixels tall. (Why not make it 100 pixels wide? Then the height would be $100 \cdot 2 / 3 \approx 67$, which seems more awkward.)

Now you can compute the coordinates of all the important points of the shape:



Step 3 Write Java statements to draw the shapes.

In our example, there are two rectangles and two lines:

```
Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
Rectangle rightRectangle = new Rectangle(160, 100, 30, 60);
Line2D.Double topLine = new Line2D.Double(130, 100, 160, 100);
Line2D.Double bottomLine = new Line2D.Double(130, 160, 160, 160);
```

If you are more ambitious, then you can express the coordinates in terms of a few variables. In the case of the flag, we have arbitrarily chosen the top-left corner and the width. All other coordinates follow from those choices. If you decide to follow the ambitious approach, then the rectangles and lines are determined as follows:

```
Rectangle leftRectangle = new Rectangle(
      xLeft, yTop,
      width / 3, width * 2 / 3);
Rectangle rightRectangle = new Rectangle(
      xLeft + 2 * width / 3, yTop,
      width / 3, width * 2 / 3);
Line2D.Double topLine = new Line2D.Double(
      xLeft + width / 3, yTop,
      xLeft + width * 2 / 3, yTop);
Line2D.Double bottomLine = new Line2D.Double(
      xLeft + width / 3, yTop + width * 2 / 3,
      xLeft + width * 2 / 3, yTop + width * 2 / 3);
```

Now you need to fill the rectangles and draw the lines. For the flag of Italy, the left rectangle is green and the right rectangle is red. Remember to switch colors before the filling and drawing operations:

```
g2.setColor(Color.GREEN);
g2.fill(leftRectangle);
g2.setColor(Color.RED);
g2.fill(rightRectangle);
g2.setColor(Color.BLACK);
g2.draw(topLine);
g2.draw(bottomLine);
```

Step 4 Combine the drawing statements with the component "plumbing".

```
public class MyComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      // Drawing instructions
      . . .
   }
}
```

In our simple example, you could add all shapes and drawing instructions inside the `paintComponent` method:

```
public class ItalianFlagComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
      . . .
      g2.setColor(Color.GREEN);
      g2.fill(leftRectangle);
      . . .
   }
}
```

That approach is acceptable for simple drawings, but it is not very object-oriented. After all, a flag is an object. It is better to make a separate class for the flag. Then you can draw different flags at different positions. Specify the sizes in a constructor and supply a `draw` method:

```
public class ItalianFlag
{
   private int xLeft;
   private int yTop;
   private int width;

   public ItalianFlag(int x, int y, int aWidth)
   {
      xLeft = x;
      yTop = y;
      width = aWidth;
   }

   public void draw(Graphics2D g2)
   {
      Rectangle leftRectangle = new Rectangle(
            xLeft, yTop,
            width / 3, width * 2 / 3);
```

```
      . . .
      g2.setColor(Color.GREEN);
      g2.fill(leftRectangle);
      . . .
   }
}
```

You still need a separate class for the component, but it is very simple:

```
public class ItalianFlagComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      ItalianFlag flag = new ItalianFlag(100, 100, 90);
      flag.draw(g2);
   }
}
```

Step 5 Write the viewer class.

Provide a viewer class, with a `main` method in which you construct a frame, add your component, and make your frame visible. The viewer class is completely routine; you only need to change a single line to show a different component.

```
public class ItalianFlagViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      frame.setSize(300, 400);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      ItalianFlagComponent component = new ItalianFlagComponent();
      frame.add(component);

      frame.setVisible(true);
   }
}
```

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download the complete flag drawing program.

CHAPTER SUMMARY

Understand instance variables and the methods that access them.

- An object's instance variables store the data required for executing its methods.
- Each object of a class has its own set of instance variables.
- Private instance variables can only be accessed by methods of the same class.
- Encapsulation is the process of hiding implementation details and providing methods for data access.

- Encapsulation allows a programmer to use a class without having to know its implementation.
- Information hiding makes it simpler for the implementor of a class to locate errors and change implementations.

Write method and constructor headers that describe the public interface of a class.

- In order to implement a class, you first need to know which methods are required.
- Constructors set the initial data for objects.
- The constructor name is always the same as the class name.
- Use documentation comments to describe the classes and public methods of your programs.
- Provide documentation comments for every class, every method, every parameter variable, and every return value.

Implement a class.

- The private implementation of a class consists of instance variables, and the bodies of constructors and methods.

Write tests that verify that a class works correctly.

- A unit test verifies that a class works correctly in isolation, outside a complete program.
- To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.

Use the technique of object tracing for visualizing object behavior.

- Write the methods on the front of a card and the instance variables on the back.
- Update the values of the instance variables when a mutator method is called.

Compare initialization and lifetime of instance, local, and parameter variables.

- Local variables are declared in the body of a method.
- When a method exits, its local variables are removed.
- Instance variables are initialized to a default value, but you must initialize local variables.

Recognize the use of the implicit parameter in method declarations.

- Use of an instance variable name in a method denotes the instance variable of the implicit parameter.
- The `this` reference denotes the implicit parameter.
- A local variable shadows an instance variable with the same name. You can access the instance variable name through the `this` reference.
- A method call without an implicit parameter is applied to the same object.

Implement classes that draw graphical shapes.

- It is a good idea to make a class for any part of a drawing that can occur more than once.
- To figure out how to draw a complex shape, make a sketch on graph paper.

REVIEW QUESTIONS

■ **R3.1** What is the public interface of the `Counter` class in Section 3.1? How does it differ from the implementation of the class?

■ **R3.2** What is encapsulation? Why is it useful?

■ **R3.3** Instance variables are a part of the hidden implementation of a class, but they aren't actually hidden from programmers who have the source code of the class. Explain to what extent the `private` reserved word provides information hiding.

■ **R3.4** Consider a class `Grade` that represents a letter grade, such as A+ or B. Give two choices of instance variables that can be used for implementing the `Grade` class.

■■ **R3.5** Consider a class `Time` that represents a point in time, such as 9 A.M. or 3:30 P.M. Give two different sets of instance variables that can be used for implementing the `Time` class.

■ **R3.6** Suppose the implementor of the `Time` class of Exercise R3.5 changes from one implementation strategy to another, keeping the public interface unchanged. What do the programmers who use the `Time` class need to do?

■■ **R3.7** You can read the `value` instance variable of the `Counter` class with the `getValue` accessor method. Should there be a `setValue` mutator method to change it? Explain why or why not.

■■ **R3.8**

a. Show that the `BankAccount(double initialBalance)` constructor is not strictly necessary. That is, if we removed that constructor from the public interface, how could a programmer still obtain `BankAccount` objects with an arbitrary balance?

b. Conversely, could we keep only the `BankAccount(double initialBalance)` constructor and remove the `BankAccount()` constructor?

■■ **R3.9** Why does the `BankAccount` class not have a `reset` method?

■ **R3.10** What happens in our implementation of the `BankAccount` class when more money is withdrawn from the account than the current balance?

■■ **R3.11** What is the `this` reference? Why would you use it?

■■ **R3.12** What does the following method do? Give an example of how you can call the method.

```
public class BankAccount
{
   public void mystery(BankAccount that, double amount)
   {
      this.balance = this.balance - amount;
      that.balance = that.balance + amount;
   }
   . . . // Other bank account methods
}
```

■■ **R3.13** Suppose you want to implement a class `TimeDepositAccount`. A time deposit account has a fixed interest rate that should be set in the constructor, together with the initial balance. Provide a method to get the current balance. Provide a method to add the earned interest to the account. This method should have no arguments because the interest rate is already known. It should have no return value because you already

provided a method for obtaining the current balance. It is not possible to deposit additional funds into this account. Provide a `withdraw` method that removes the entire balance. Partial withdrawals are not allowed.

■ **R3.14** Consider the following implementation of a class `Square`:

```
public class Square
{
   private int sideLength;
   private int area; // Not a good idea

   public Square(int length)
   {
      sideLength = length;
   }

   public int getArea()
   {
      area = sideLength * sideLength;
      return area;
   }
}
```

Why is it not a good idea to introduce an instance variable for the area? Rewrite the class so that `area` is a local variable.

■■ **R3.15** Consider the following implementation of a class `Square`:

```
public class Square
{
   private int sideLength;
   private int area;

   public Square(int initialLength)
   {
      sideLength = initialLength;
      area = sideLength * sideLength;
   }

   public int getArea() { return area; }
   public void grow() { sideLength = 2 * sideLength; }
}
```

What error does this class have? How would you fix it?

■■ **Testing R3.16** Provide a unit test class for the `Counter` class in Section 3.1.

■■ **Testing R3.17** Read Exercise E3.9, but do not implement the `Car` class yet. Write a tester class that tests a scenario in which gas is added to the car, the car is driven, more gas is added, and the car is driven again. Print the actual and expected amount of gas in the tank.

■ **R3.18** Using the object tracing technique described in Section 3.5, trace the program at the end of Section 3.4.

■■ **R3.19** Using the object tracing technique described in Section 3.5, trace the program in How To 3.1.

■■ **R3.20** Using the object tracing technique described in Section 3.5, trace the program in Worked Example 3.1.

■■■ **R3.21** Design a modification of the `BankAccount` class in which the first five transactions per month are free and a $1 fee is charged for every additional transaction. Provide a method that deducts the fee at the end of a month. What additional instance variables do you need? Using the object tracing technique described in Section 3.5, trace a scenario that shows how the fees are computed over two months.

■■ **Graphics R3.22** Suppose you want to extend the car viewer program in Section 3.8 to show a suburban scene, with several cars and houses. Which classes do you need?

■■■ **Graphics R3.23** Explain why the calls to the `getWidth` and `getHeight` methods in the `CarComponent` class have no explicit parameter.

■■ **Graphics R3.24** How would you modify the `Car` class in order to show cars of varying sizes?

PRACTICE EXERCISES

■ **E3.1** We want to add a button to the tally counter in Section 3.1 that allows an operator to undo an accidental button click. Provide a method

```
public void undo()
```

that simulates such a button. As an added precaution, make sure that clicking the undo button more often than the click button has no effect. (*Hint:* The call `Math.max(n, 0)` returns `n` if `n` is greater than zero, zero otherwise.)

■ **E3.2** Simulate a tally counter that can be used to admit a limited number of people. First, the limit is set with a call

```
public void setLimit(int maximum)
```

If the click button is clicked more often than the limit, it has no effect. (*Hint:* The call `Math.min(n, limit)` returns `n` if `n` is less than `limit`, and `limit` otherwise.).

■ **Testing E3.3** Write a `BankAccountTester` class whose `main` method constructs a bank account, deposits $1,000, withdraws $500, withdraws another $400, and then prints the remaining balance. Also print the expected result.

■ **E3.4** Add a method

```
public void addInterest(double rate)
```

to the `BankAccount` class that adds interest at the given rate. For example, after the statements

```
BankAccount momsSavings = new BankAccount(1000);
momsSavings.addInterest(10); // 10 percent interest
```

the balance in `momsSavings` is $1,100. Also supply a `BankAccountTester` class that prints the actual and expected balance.

■ **E3.5** Write a class `SavingsAccount` that is similar to the `BankAccount` class, except that it has an added instance variable `interest`. Supply a constructor that sets both the initial balance and the interest rate. Supply a method `addInterest` (with no explicit parameter) that adds interest to the account. Write a `SavingsAccountTester` class that constructs a savings account with an initial balance of $1,000 and an interest rate of 10 percent. Then apply the `addInterest` method and print the resulting balance. Also compute the expected result by hand and print it.

■■■ **E3.6** Add a method `printReceipt` to the `CashRegister` class. The method should print the prices of all purchased items and the total amount due. *Hint:* You will need to form a string of all prices. Use the `concat` method of the `String` class to add additional items to that string. To turn a price into a string, use the call `String.valueOf(price)`.

■ **E3.7** After closing time, the store manager would like to know how much business was transacted during the day. Modify the `CashRegister` class to enable this functionality. Supply methods `getSalesTotal` and `getSalesCount` to get the total amount of all sales and the number of sales. Supply a method `reset` that resets any counters and totals so that the next day's sales start from zero.

■■ **E3.8** Implement a class `Employee`. An employee has a name (a string) and a salary (a `double`). Provide a constructor with two arguments

```
public Employee(String employeeName, double currentSalary)
```

and methods

```
public String getName()
public double getSalary()
public void raiseSalary(double byPercent)
```

These methods return the name and salary, and raise the employee's salary by a certain percentage. Sample usage:

```
Employee harry = new Employee("Hacker, Harry", 50000);
harry.raiseSalary(10); // Harry gets a 10 percent raise
```

Supply an `EmployeeTester` class that tests all methods.

■■ **E3.9** Implement a class `Car` with the following properties. A car has a certain fuel efficiency (measured in miles/gallon or liters/km—pick one) and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is 0. Supply a method `drive` that simulates driving the car for a certain distance, reducing the amount of gasoline in the fuel tank. Also supply methods `getGasInTank`, returning the current amount of gasoline in the fuel tank, and `addGas`, to add gasoline to the fuel tank. Sample usage:

```
Car myHybrid = new Car(50); // 50 miles per gallon
myHybrid.addGas(20); // Tank 20 gallons
myHybrid.drive(100); // Drive 100 miles
double gasLeft = myHybrid.getGasInTank(); // Get gas remaining in tank
```

You may assume that the `drive` method is never called with a distance that consumes more than the available gas. Supply a `CarTester` class that tests all methods.

■ **E3.10** Implement a class `Product`. A product has a name and a price, for example `new Product("Toaster", 29.95)`. Supply methods `getName`, `getPrice`, and `reducePrice`. Supply a program `ProductPrinter` that makes two products, prints the name and price, reduces their prices by $5.00, and then prints the prices again.

■■ **E3.11** Provide a class for authoring a simple letter. In the constructor, supply the names of the sender and the recipient:

```
public Letter(String from, String to)
```

Supply a method

```
public void addLine(String line)
```

to add a line of text to the body of the letter.

Supply a method

```
public String getText()
```

that returns the entire text of the letter. The text has the form:

```
Dear recipient name:
blank line
first line of the body
second line of the body
. . .
last line of the body
blank line
Sincerely,
blank line
sender name
```

Also supply a class `LetterPrinter` that prints this letter.

```
Dear John:

I am sorry we must part.
I wish you all the best.

Sincerely,

Mary
```

Construct an object of the `Letter` class and call `addLine` twice.

Hints: (1) Use the `concat` method to form a longer string from two shorter strings. (2) The special string `"\n"` represents a new line. For example, the statement

```
body = body.concat("Sincerely,").concat("\n");
```

adds a line containing the string `"Sincerely,"` to the body.

■■ **E3.12** Write a class `Bug` that models a bug moving along a horizontal line. The bug moves either to the right or left. Initially, the bug moves to the right, but it can turn to change its direction. In each move, its position changes by one unit in the current direction. Provide a constructor

```
public Bug(int initialPosition)
```

and methods

```
public void turn()
public void move()
public int getPosition()
```

Sample usage:

```
Bug bugsy = new Bug(10);
bugsy.move(); // Now the position is 11
bugsy.turn();
bugsy.move(); // Now the position is 10
```

Your `BugTester` should construct a bug, make it move and turn a few times, and print the actual and expected position.

■■ **E3.13** Implement a class `Moth` that models a moth flying along a straight line. The moth has a position, which is the distance from a fixed origin. When the moth moves toward a point of light, its new position is halfway between its old position and the position of the light source. Supply a constructor

```
public Moth(double initialPosition)
```

and methods

```
public void moveToLight(double lightPosition)
public double getPosition()
```

Your `MothTester` should construct a moth, move it toward a couple of light sources, and check that the moth's position is as expected.

■■■ **Graphics E3.14** Write a program that fills the window with a large ellipse, with a black outline and filled with your favorite color. The ellipse should touch the window boundaries, even if the window is resized. Call the `getWidth` and `getHeight` methods of the `JComponent` class in the `paintComponent` method.

■■ **Graphics E3.15** Draw a shooting target—a set of concentric rings in alternating black and white colors. *Hint:* Fill a black circle, then fill a smaller white circle on top, and so on. Your program should be composed of classes `Target`, `TargetComponent`, and `TargetViewer`.

■■ **Graphics E3.16** Write a program that draws a picture of a house. It could be as simple as the accompanying figure, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever). Implement a class `House` and supply a method `draw(Graphics2D g2)` that draws the house.

■■ **Graphics E3.17** Extend Exercise E3.16 by supplying a `House` constructor for specifying the position and size. Then populate your screen with a few houses of different sizes.

■■ **Graphics E3.18** Change the car viewer program in Section 3.8 to make the cars appear in different colors. Each `Car` object should store its own color. Supply modified `Car` and `CarComponent` classes.

■■ **Graphics E3.19** Change the `Car` class so that the size of a car can be specified in the constructor. Change the `CarComponent` class to make one of the cars appear twice the size of the original example.

■■ **Graphics E3.20** Write a program to plot the string "HELLO", using only lines and circles. Do not call `drawString`, and do not use `System.out`. Make classes `LetterH`, `LetterE`, `LetterL`, and `LetterO`.

■■ **Graphics E3.21** Write a program that displays the Olympic rings. Color the rings in the Olympic colors. Provide classes `OlympicRing`, `OlympicRingViewer` and `OlympicRingComponent`.

■■ **Graphics E3.22** Make a bar chart to plot the following data set. Label each bar. Make the bars horizontal for easier labeling. Provide a class `BarChartViewer` and a class `BarChartComponent`.

Bridge Name	Longest Span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinac	3,800

PROGRAMMING PROJECTS

■ **P3.1** Enhance the `CashRegister` class so that it counts the purchased items. Provide a `getItemCount` method that returns the count.

■■■ **P3.2** Support computing sales tax in the `CashRegister` class. The tax rate should be supplied when constructing a `CashRegister` object. Add `recordTaxablePurchase` and `getTotalTax` methods. (Amounts added with `recordPurchase` are not taxable.) The `giveChange` method should correctly reflect the sales tax that is charged on taxable items.

■■ **P3.3** Implement a class `Balloon`. A balloon starts out with radius 0. Supply a method

```
public void inflate(double amount)
```

that increases the radius by the given amount. Supply a method

```
public double getVolume()
```

that returns the current volume of the balloon. Use `Math.PI` for the value of π. To compute the cube of a value `r`, just use `r * r * r`.

■■■ **P3.4** Implement a class `Student`. For the purpose of this exercise, a student has a name and a total quiz score. Supply an appropriate constructor and methods `getName()`, `addQuiz(int score)`, `getTotalScore()`, and `getAverageScore()`. To compute the average, you also need to store the *number of quizzes* that the student took.

Supply a `StudentTester` class that tests all methods.

■ **P3.5** Write a class `Battery` that models a rechargeable battery. A battery has a constructor

```
public Battery(double capacity)
```

where capacity is a value measured in milliampere hours. A typical AA battery has a capacity of 2000 to 3000 mAh. The method

```
public void drain(double amount)
```

drains the capacity of the battery by the given amount. The method

```
public void charge()
```

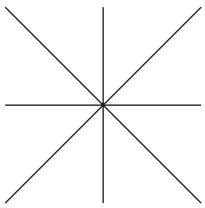
charges the battery to its original capacity.

The method

```
public double getRemainingCapacity()
```

gets the remaining capacity of the battery.

■■ **Graphics P3.6** Write a program that draws three stars. Use classes `Star`, `StarComponent`, and `StarViewer`. Each star should look like this:



■■ **P3.7** Implement a class `RoachPopulation` that simulates the growth of a roach population. The constructor takes the size of the initial roach population. The `breed` method simulates a period in which the roaches breed, which doubles their population. The

`spray(double percent)` method simulates spraying with insecticide, which reduces the population by the given percentage. The `getRoaches` method returns the current number of roaches. A program called `RoachSimulation` simulates a population that starts out with 10 roaches. Breed, spray to reduce the population by 10 percent, and print the roach count. Repeat three more times.

■■ **P3.8** Implement a `VotingMachine` class that can be used for a simple election. Have methods to clear the machine state, to vote for a Democrat, to vote for a Republican, and to get the tallies for both parties.

■■■ **P3.9** In this project, you will enhance the `BankAccount` class and see how abstraction and encapsulation enable evolutionary changes to software.

Begin with a simple enhancement: charging a fee for every deposit and withdrawal. Supply a mechanism for setting the fee and modify the `deposit` and `withdraw` methods so that the fee is levied. Test your resulting class and check that the fee is computed correctly.

Now make a more complex change. The bank will allow a fixed number of free transactions (deposits or withdrawals) every month, and charge for transactions exceeding the free allotment. The charge is not levied immediately but at the end of the month.

Supply a new method `deductMonthlyCharge` to the `BankAccount` class that deducts the monthly charge and resets the transaction count. (*Hint:* Use `Math.max(actual transaction count, free transaction count)` in your computation.)

Produce a test program that verifies that the fees are calculated correctly over several months.

■■■ **P3.10** In this project, you will explore an object-oriented alternative to the "Hello, World" program in Chapter 1.

Begin with a simple `Greeter` class that has a single method, `sayHello`. That method should *return* a string, not print it. Create two objects of this class and invoke their `sayHello` methods. Of course, both objects return the same answer.

Enhance the `Greeter` class so that each object produces a customized greeting. For example, the object constructed as `new Greeter("Dave")` should say `"Hello, Dave"`. (Use the `concat` method to combine strings to form a longer string, or peek ahead at Section 4.5 to see how you can use the `+` operator for the same purpose.)

Add a method `sayGoodbye` to the `Greeter` class.

Finally, add a method `refuseHelp` to the `Greeter` class. It should return a string such as `"I am sorry, Dave. I am afraid I can't do that."`

If you use BlueJ, place two `Greeter` objects on the workbench (one that greets the world and one that greets Dave) and invoke methods on them. Otherwise, write a tester program that constructs these objects, invokes methods, and prints the results.

ANSWERS TO SELF-CHECK QUESTIONS

1.
```
public void unclick()
{
   value = value - 1;
}
```

2. You can only access them by invoking the methods of the `Clock` class.

3. In one of the methods of the `Counter` class.

4. The programmers who designed and implemented the Java library.

5. Other programmers who work on the personal finance application.

6.
```
harrysChecking.withdraw(
   harrysChecking.getBalance())
```

7. The `withdraw` method has return type `void`. It doesn't return a value. Use the `getBalance` method to obtain the balance after the withdrawal.

8. Add an `accountNumber` parameter variable to the constructors, and add a `getAccountNumber` method. There is no need for a `setAccountNumber` method—the account number never changes after construction.

9.
```
/**
   Constructs a new bank account with a given
   initial balance.
   @param accountNumber the account number for
   this account
   @param initialBalance the initial balance for
   this account
*/
```

10. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table.

11. An instance variable needs to be added to the class:
```
private int accountNumber;
```

12. Because the `balance` instance variable is accessed from the `main` method of `BankRobber`. The compiler will report an error because `main` is not a method of the `BankAccount` class and has no access to `BankAccount` instance variables.

13.
```
public int getWidth()
{
   return width;
}
```

14. There is more than one correct answer. One possible implementation is as follows:
```
public void translate(int dx, int dy)
{
   int newx = x + dx;
   x = newx;
   int newy = y + dy;
   y = newy;
}
```

15. One `BankAccount` object, no `BankAccountTester` object. The purpose of the `BankAccountTester` class is merely to hold the `main` method.

16. In those environments, you can issue interactive commands to construct `BankAccount` objects, invoke methods, and display their return values.

17.

front

gasLeft	milesPerGallon
0	25

back

18.

gasLeft	milesPerGallon
~~0~~	25
~~20~~	
~~16~~	
~~8~~	
13	

19.

gasLeft	milesPerGallon	totalMiles
0	25	0

20.

gasLeft	milesPerGallon	totalMiles
~~0~~ ~~20~~ ~~16~~ ~~8~~ 13	25	0 100 300

21. Variables of both categories belong to methods—they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the values supplied as arguments in the call; local variables must be explicitly initialized.

22. After computing the change due, `payment` and `purchase` were set to zero. If the method returned `payment - purchase`, it would always return zero.

23.

reg1.purchase	reg1.payment	change
~~19.5~~	~~20~~	
		0.5
0	0	
		X

24. One implicit parameter, called `this`, of type `BankAccount`, and one explicit parameter, called `amount`, of type `double`.

25. It is not a legal expression. `this` is of type `BankAccount` and the `BankAccount` class has no instance variable named `amount`.

26. No implicit parameter—the `main` method is not invoked on any object—and one explicit parameter, called `args`.

27. `CarComponent`

28. In the `draw` method of the `Car` class, call

```
g2.fill(frontTire);
g2.fill(rearTire);
```

29. Double all measurements in the `draw` method of the `Car` class.