33

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 parameters and return values

T To implement test programs

- To be able to browse the API documentation
- To realize the difference between objects and object references

G To write programs that display simple shapes

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 define *classes* that describe the behavior of these objects. In this chapter, you will learn how to manipulate objects that belong to predefined classes. This knowledge will prepare you for the next chapter in which you will learn how to implement your own classes.

33

34

2.1 Types and Variables

In Java, every value has a *type*. For example, `"Hello, World"` has the type `String`, the object `System.out` has the type `PrintStream`, and the number 13 has the type `int` (an abbreviation for "integer"). The type tells you what you can do with the values. You can call `println` on any object of type `PrintStream`. You can compute the sum or product of any two integers.

In Java, every value has a type.

You often want to store values so that you can use them at a later time. To remember an object, you need to hold it in a *variable*. A variable is a storage location in the computer's memory that has a *type*, a *name*, and a contents. For example, here we declare three variables:

```
String greeting = "Hello, World!";
PrintStream printer = System.out;
int luckyNumber = 13;
```

The first variable is called `greeting`. It can be used to store `String` values, and it is set to the value `"Hello, World!"`. The second variable stores a `PrintStream` value, and the third stores an integer.

You use variables to store values that you want to use at a later time.

Variables can be used in place of the objects that they store:

```
printer.println(greeting); // Same as System.out.println("Hello, World!")
printer.println(luckyNumber); // Same as System.out.println(13)
```

34

35

SYNTAX 2.1 Variable Definition

typeName variableName = value;

or

typeName variableName;

Example:

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

Purpose:

To define a new variable of a particular type and optionally supply an initial value

When you declare your own variables, you need to make two decisions.

- What type should you use for the variable?
- What name should you give the variable?

The type depends on the intended use. If you need to store a string, use the `String` type for your variable.

It is an error to store a value whose class does not match the type of the variable. For example, the following is an error:

```
String greeting = 13; // ERROR: Types don't match
```

You cannot use a `String` variable to store an integer. The compiler checks type mismatches to protect you from errors.

When deciding on a name for a variable, you should make a choice that describes the purpose of the variable. For example, the variable name `greeting` is a better choice than the name `g`.

Identifiers for variables, methods, and classes are composed of letters, digits, and underscore characters.

An *identifier* is the name of a variable, method, or class. Java imposes the following rules for identifiers:

- Identifiers can be made up of letters, digits, and the underscore (_) and dollar sign ($) characters. They cannot start with a digit, though. For example, `greeting1` is legal but `1greeting` is not.
- You cannot use other symbols such as? or `%`. For example, `hello!` is not a legal identifier.
- Spaces are not permitted inside identifiers. Therefore, `lucky number` is not legal.
- Furthermore, you cannot use *reserved words*, such as `public`, as names; these words are reserved exclusively for their special Java meanings.

- Identifiers are also *case sensitive*; that is, `greeting` and `Greeting` are *different*.

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

These are firm rules of the Java language. If you violate one of them, the compiler will report an error. Moreover, there are a couple of *conventions* that you should follow so that other programmers will find your programs easy to read:

35

36

- Variable and method names should start with a lowercase letter. It is OK to use an occasional uppercase letter, such as `luckyNumber`. This mixture of lowercase and uppercase letters is sometimes called "camel case" because the uppercase letters stick out like the humps of a camel.
- Class names should start with an uppercase letter. For example, `Greeting` would be an appropriate name for a class, but not for a variable.

If you violate these conventions, the compiler won't complain, but you will confuse other programmers who read your code.

SELF CHECK

1. What is the type of the values 0 and "0"?
2. Which of the following are legal identifiers?

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

3. Define a variable to hold your name. Use camel case in the variable name.

2.2 The Assignment Operator

You can change the value of an existing variable with the assignment operator (=). For example, consider the variable definition

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

```
int luckyNumber = 13;
```

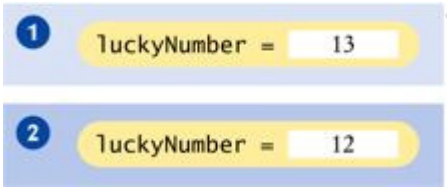
If you want to change the value of the variable, simply assign the new value:

```
luckyNumber = 12;
```

The assignment replaces the original value of the variable (see Figure 1).

In the Java programming language, the = operator denotes an *action*, to replace the value of a variable. This usage differs from the traditional usage of the = symbol, as a statement about equality.

Figure 1

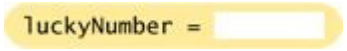


Assigning a New Value to a Variable

36

37

Figure 2



An Uninitialized Object Variable

It is an error to use a variable that has never had a value assigned to it. For example, the sequence of statements

```
int luckyNumber;
System.out.println(luckyNumber);      // ERROR—uninitialized variable
```

is an error. The compiler will complain about an "uninitialized variable" when you use a variable that has never been assigned a value. (See Figure 2.)

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

All variables must be initialized before you access them.

```
int luckyNumber;
luckyNumber = 13;
System.out.println(luckyNumber); // OK
```

Or, even better, initialize the variable when you define it.

```
int luckyNumber = 13;
System.out.println(luckyNumber); // OK
```

SYNTAX 2.2 Assignment

variableName = *value*;

Example:

```
luckyNumber = 12;
```

Purpose:

To assign a new value to a previously defined variable

SELF CHECK

4. Is `12 = 12` a valid expression in the Java language?
5. How do you change the value of the `greeting` variable to `"Hello, Nina!";?`

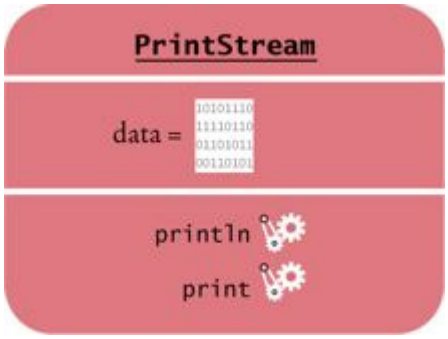
2.3 Objects, Classes, and Methods

An *object* is an entity that you can manipulate in your program. You don't usually know how the object is organized internally. However, the object has well-defined behavior, and that is what matters to us when we use it.

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

You manipulate an object by calling one or more of its *methods*. A method consists of a sequence of instructions that accesses the internal data. When you call the method, you do not know exactly what those instructions are, but you do know the purpose of the method.

Figure 3



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 3 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 Chapter 1, you encountered two objects:

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

These objects belong to different *classes*. The `System.out` object belongs to the class `PrintStream`. The `"Hello, World!"` object belongs to the class `String`. A class specifies the methods that you can apply to its objects.

You can use the `println` method with any object that belongs to the `PrintStream` class. `System.out` is one such object. It is possible to obtain other objects of the `PrintStream` class. For example, you can construct a `PrintStream` object to send output to a file. However, we won't discuss files until Chapter 11.

A class defines the methods that you can apply to its objects.

Just as the `PrintStream` class provides methods such as `println` and `print` for its objects, 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 n = greeting.length();
```

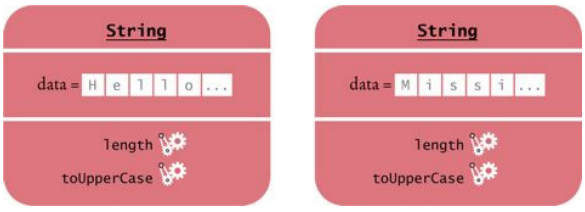
sets `n` to the number of characters in the `String` object `"Hello, World!"`. After the instructions in the length method are executed, `n` is set to 13. (The quotation marks are not part of the string, and the `length` method does not count them.)

The `length` method—unlike the `println` method—requires no input inside the parentheses. However, the `length` method yields an output, namely the character count.

38

39

Figure 4



A Representation of Two `String` Objects

In the next section, you will see in greater detail how to supply method inputs and obtain method outputs.

Let us 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"`.

When you apply a method to an object, you must make sure that the method is defined in the appropriate class. For example, it is an error to call

```
System.out.length(); // This method call is an error
```

The `PrintStream` class (to which `System.out` belongs) has no `length` method.

Let us summarize. In Java, *every object belongs to a class. The class defines the methods for the objects*. For example, the `String` class defines the `length` and `toUpperCase` methods (as well as other methods—you will learn about most of them in Chapter 4). The methods form the *public interface* of the class, telling you what you can do with the objects of the class. A class also defines 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 public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.

Figure 4 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 interface that is specified by the `String` class.

Self Check

6. How can you compute the length of the string `"Mississippi"`?

7. How can you print out the uppercase version of "`Hello, World!`"?

8. Is it legal to call `river.println()`? Why or why not?

39

40

2.4 Method Parameters and Return Values

In this section, we will examine how to provide inputs into a method, and how to obtain the output of the method.

Some methods require inputs that give details about the work that they need to do. For example, the `println` method has an input: the string that should be printed. Computer scientists use the technical term *parameter* for method inputs. We say that the string `greeting` is a parameter of the method call

A parameter is an input to a method.

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

Figure 5 illustrates passing of the parameter to the method.

Technically speaking, the `greeting` parameter is an *explicit parameter* of the `println` method. The object on which you invoke the method is also considered a parameter of the method call, called the *implicit parameter*. For example, `System.out` is the implicit parameter of the method call

The implicit parameter of a method call is the object on which the method is invoked.

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

Some methods require multiple explicit parameters, others don't require any explicit parameters at all. An example of the latter is the `length` method of the `String` class (see Figure 6). All the information that the `length` method requires to do its job—namely, the character sequence of the string—is stored in the implicit parameter object.

The `length` method differs from the `println` method in another way: it has an output. We say that the method *returns a value*, namely the number of characters in the string. You can store the return value in a variable:

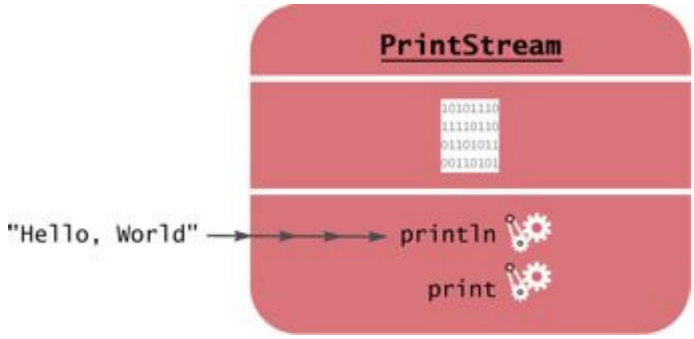
> The return value of a method is a result that the method has computed for use by the code that called it.

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

You can also use the return value as a parameter of another method:

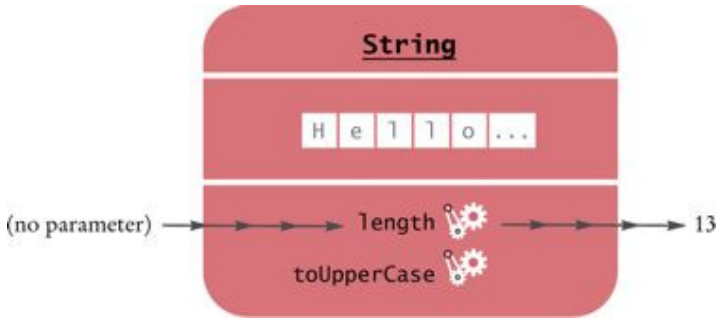
```
System.out.println(greeting.length());
```

Figure 5



Passing a Parameter to the `println` Method

Figure 6



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

The method call `greeting.length()` returns a value—the integer 13. The return value becomes a parameter of the `println` method. Figure 7 shows the process.

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

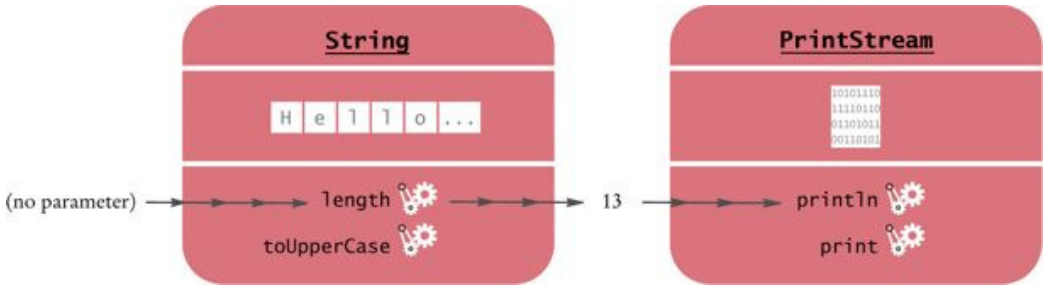
```
river.replace("issipp", "our")
```

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"` (which you can save in a variable or pass to another method).

As Figure 8 shows, this method call has

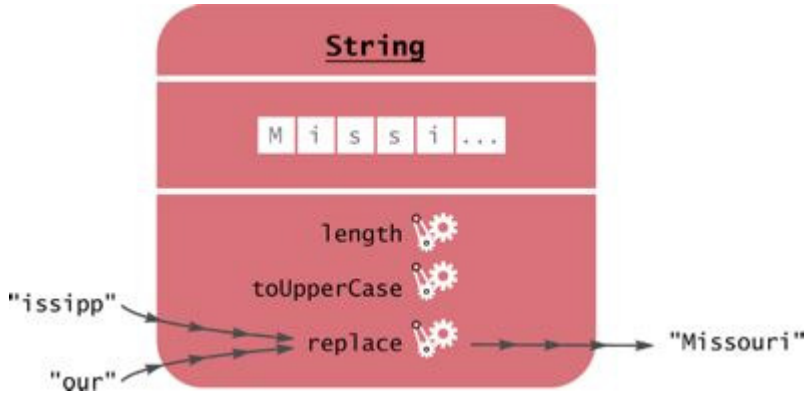
- one implicit parameter: the string `"Mississippi"`
- two explicit parameters: the strings `"issipp"` and `"our"`
- a return value: the string `"Missouri"`

Figure 7



Passing the Result of a Method Call to Another Method

Figure 8



Calling the `replace` Method

When a method is defined in a class, the definition specifies the types of the explicit parameters and the return value. For example, the `String` class defines the `length` method as

```
public int length()
```

That is, there are no explicit parameters, and the return value has the type `int`. (For now, all the methods that we consider will be "public" methods—see Chapter 10 for more restricted methods.)

The type of the implicit parameter is the class that defines the method—`String` in our case. It is not mentioned in the method definition—hence the term "implicit".

The `replace` method is defined as

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

To call the `replace` method, you supply two explicit parameters, `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 defines the `println` method as

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

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

A method name is overloaded if a class has more than one method with the same name (but different parameter types).

```
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

9. What are the implicit parameters, explicit parameters, and return values in the method call `river.length()`?
10. What is the result of the call `river.replace("p", "s")`?
11. What is the result of the call `greeting.replace("World", "Dave").length()`?
12. How is the `toUpperCase` method defined in the `String` class?

42
43

2.5 Number Types

Java has separate types for *integers* and *floating-point numbers*. Integers are whole numbers; floating-point numbers can have fractional parts. For example, 13 is an integer and 1.3 is a floating-point number.

The double type denotes floating-point numbers that can have fractional parts.

The name "floating-point" describes the representation of the number in the computer as a sequence of the significant digits and an indication of the position of the decimal point. For example, the numbers 13000, 1.3, 0.00013 all have the same decimal digits: 13. When a floating-point number is multiplied or divided by 10, only the position of the decimal point changes; it "floats". This representation is related to the "scientific"

notation 1.3×10^{-4}. (Actually, the computer represents numbers in base 2, not base 10, but the principle is the same.)

If you need to process numbers with a fractional part, you should use the type called `double`, which stands for "double precision floating-point number". Think of a number in `double` format as any number that can appear in the display panel of a calculator, such as 1.3 or –0.333333333.

Do not use commas when you write numbers in Java. For example, 13,000 must be written as `13000`. To write numbers in exponential notation in Java, use the notation `E`*n* instead of "$\times 10^n$". For example, 1.3×10^{-4} is written as `1.3E-4`.

You may wonder why Java has separate integer and floating-point number types. Pocket calculators don't need a separate integer type; they use floating-point numbers for all calculations. However, integers have several advantages over floating-point numbers. They take less storage space, are processed faster, and don't cause rounding errors. You will want to use the `int` type for quantities that can never have fractional parts, such as the length of a string. Use the `double` type for quantities that can have fractional parts, such as a grade point average.

There are several other number types in Java that are not as commonly used. We will discuss these types in Chapter 4. For most practical purposes, however, the `int` and `double` types are all you need for processing numbers.

In Java, the number types (`int, double`, and the less commonly used types) are *primitive types*, not classes. Numbers are not objects. The number types have no methods.

> In Java, numbers are not objects and number types are not classes.

However, you can combine numbers with operators such as + and –, as in `10 + n` or `n - 1`. To multiply two numbers, use the `*` operator. For example, $10 \times n$ is written as `10 * n`.

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

As in mathematics, the `*` operator binds more strongly than the `+` operator. That is, `x + y * 2` means the sum of `x` and `y * 2`. If you want to multiply the sum of `x` and `y` with 2, use parentheses:

```
(x + y) * 2
```


SELF CHECK

13. Which number type would you use for storing the area of a circle?
14. Why is the expression `13.println()` an error?
15. Write an expression to compute the average of the values `x` and `y`.

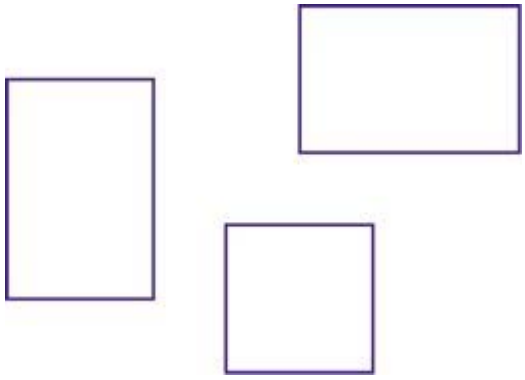
2.6 Constructing Objects

Most Java programs will want to work on a variety of objects. In this section, you will see how to *construct* new objects. This allows you to go beyond `String` objects and the predefined `System.out` object.

To learn about object construction, let us turn to another class: the `Rectangle` class in the Java class library. Objects of type `Rectangle` describe rectangular shapes—see Figure 9. 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.

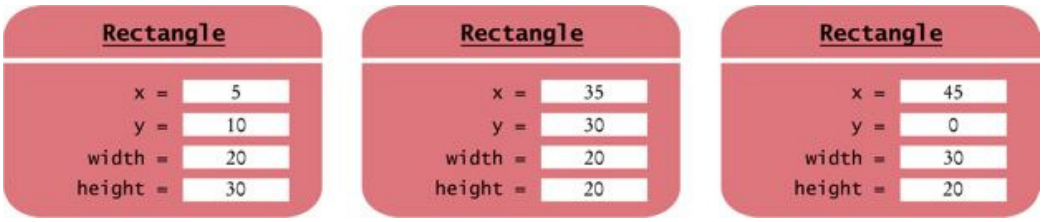
Note that a `Rectangle` object isn't a rectangular shape—it is an object that contains a set of numbers. The numbers *describe* the rectangle (see Figure 10). Each rectangle is described by the *x*- and *y*-coordinates of its top-left corner, its width, and its height.

Figure 9



Rectangular Shapes

Figure 10



`Rectangle` Objects

44

45

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 parameters, 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 parameters that are 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 parameters (in this case, 5, 10, 20, and 30) to initialize the data of the object.
3. It returns the object.

Usually the output of the `new` operator is stored in a variable. For example,

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

The process of creating a new object is called *construction*. The four values 5, 10, 20, and 30 are called the *construction parameters*. Note that the `new` expression is *not* a complete statement. You use the value of a `new` expression just like a method return value: Assign it to a variable or pass it to another method.

Some classes let you construct objects in multiple ways. For example, you can also obtain a `Rectangle` object by supplying no construction parameters 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.

SYNTAX 2.3 Object Construction

new *ClassName(parameters)*

Example:

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

Purpose:

To construct a new object, initialize it with the construction parameters, and return a reference to the constructed object

Self Check

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

17. What does the following statement print?

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

Common Error 2.1: 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.

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

2.7 Accessor and Mutator Methods

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 state of an object is called a *mutator* method.

An accessor method does not change the state of its implicit parameter. A mutator method changes the state.

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);
```

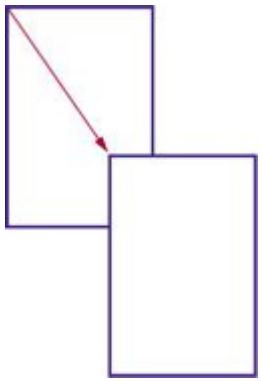
moves the rectangle by 15 units in the *x*-direction and 25 units in the *y*-direction (see Figure 11). Moving a rectangle doesn't change its width or height, but it changes the top-left corner. Afterwards, the top-left corner is at (20, 35).

This method is a mutator because it modifies the implicit parameter object.

46

47

Figure 11



Using the `translate` Method to Move a Rectangle

Self Check

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

19. Which call to `translate` is needed to move the `box` rectangle so that its top-left corner is the origin (0, 0)?

2.8 Implementing a Test Program

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 activity. When you implement your own methods, you should always supply programs to test them.

In this book, we use a very simple format for test programs. You will now see such a test program that tests a method in the `Rectangle` class. The program performs the following 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.

Whenever you write a program to test your own classes, you need to follow these steps as well.

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 `Rectangle-Tester` class).

47

48

```
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 what value we expect to see.

This is a very important step. You want to spend some time thinking what the expected result is 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.

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

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

Here is a complete program that tests the moving of a rectangle.

ch02/rectangle/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 }
```

Output

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

48

49

For this program, we needed to carry out another step: We needed to *import* the `Rectangle` class from a *package*. A package is a collection of classes with a related purpose. All classes in the standard library are contained in packages. 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.

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

To use the `Rectangle` class from the `java.awt` 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

```
import packageName.ClassName;
```

Example:

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

Purpose

To import a class from a package for use in a program

Self Check

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

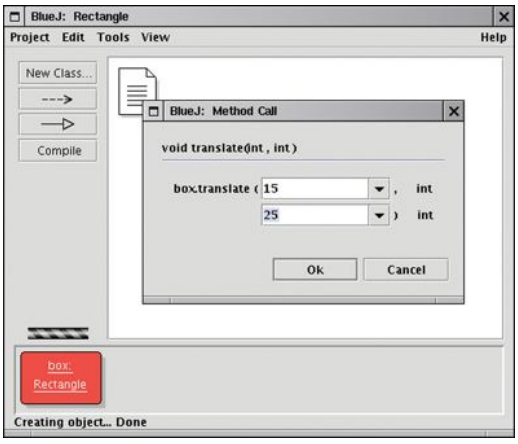
21. Why doesn't the `MoveTester` program need to print the width and height of the rectangle?

22. The `Random` class is defined in the `java.util` package. What do you need to do in order to use that class in your program?

Advanced 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 Testing a Method Call in BlueJ) 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 [1]. Another excellent environment for interactively exploring objects is Dr. Java [2].

49
50



Testing a Method Call in BlueJ

2.9 The API Documentation

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*.

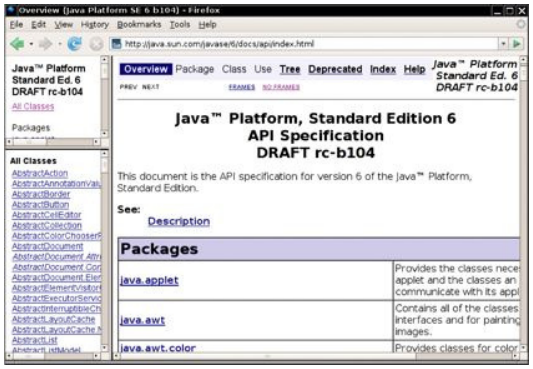
You can find the API documentation on the Web [3]. Point your web browser to `http://java.sun.com/javase/6/docs/api/index.html`. Alternatively, you can download and install the API documentation onto your own computer—see Productivity Hint 2.1.

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

The API documentation documents all classes in the Java library—there are thousands of them (see Figure 12). 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).

Figure 12



The API Documentation of the Standard Java Library

Figure 13

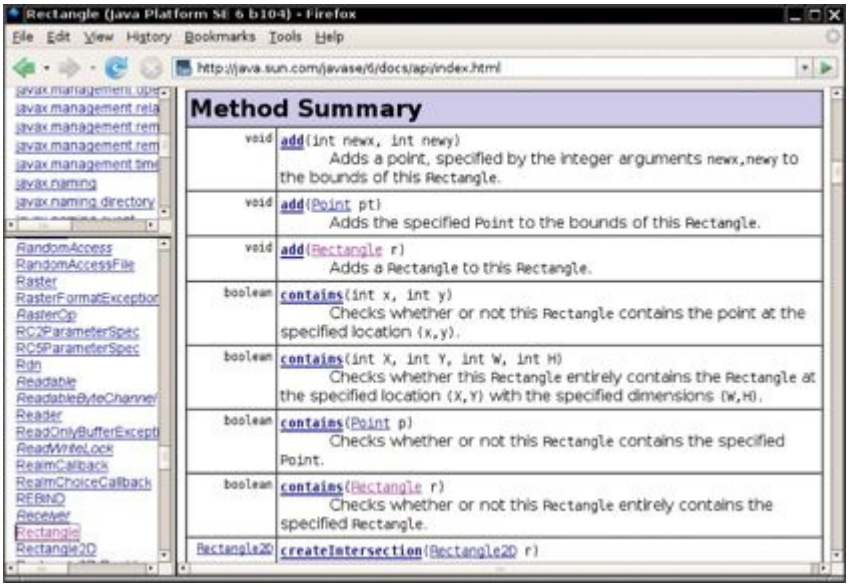
The API Documentation for the `Rectangle` Class

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). Click on the link of a method to get a detailed description (see Figure 15).

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.

51

Figure 14



The Method Summary for the `Rectangle` Class

Figure 15

The API Documentation of the `translate` Method

Appendix C contains an abbreviated version of the API documentation. You may find the abbreviated documentation easier to use than the full documentation. It is fine if you rely on the abbreviated documentation for your first programs, but you should eventually move on to the real thing.

52

53

Self Check

23. 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!"`?

24. 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.)

Productivity Hint 2.1: 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. Since 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://java.sun.com/javase/downloads/index.html`.

2.10 Object References

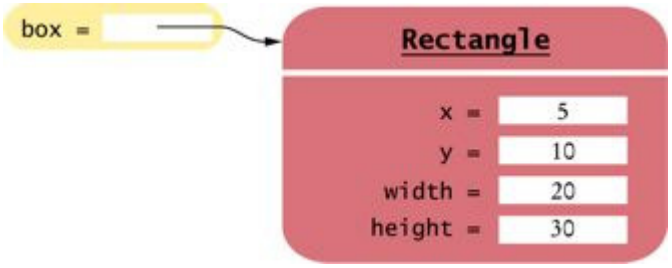
In Java, a variable whose type is a class does not actually hold an object. It merely holds the memory *location* of an object. The object itself is stored elsewhere—see Figure 16.

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

An object reference describes the location of an object.

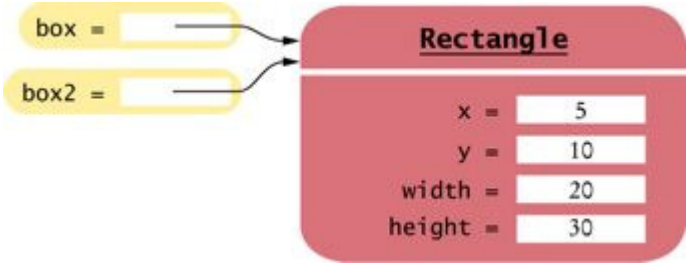
```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Figure 16



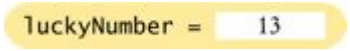
An Object Variable Containing an Object Reference

Figure 17



Two Object Variables Referring to the Same Object

Figure 18



A Number Variable Stores a Number

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. You can have two object variables refer to the same object:

```
Rectangle box2 = box;
```

Now you can access the same `Rectangle` object both as `box` and as `box2`, as shown in Figure 17.

> Multiple object variables can contain references to the same object.

However, number variables actually store numbers. When you define

```
int luckyNumber = 13;
```

then the `luckyNumber` variable holds the number 13, not a reference to the number (see Figure 18).

You can see the difference between number variables and object variables when you make a copy of a variable. When you copy a primitive type value, 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.

> Number variables store numbers. Object variables store references.

Consider the following code, which copies a number and then changes the copy (see Figure 19):

```
int luckyNumber = 13;
int luckyNumber2 = luckyNumber;
luckyNumber2 = 12;
```

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

Now consider the seemingly analogous code with `Rectangle` objects.

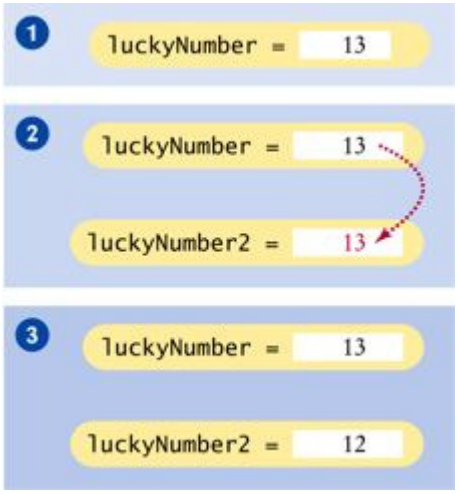
```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

```
Rectangle box2 = box; // See Figure 20
box2.translate(15, 25);
```

54

55

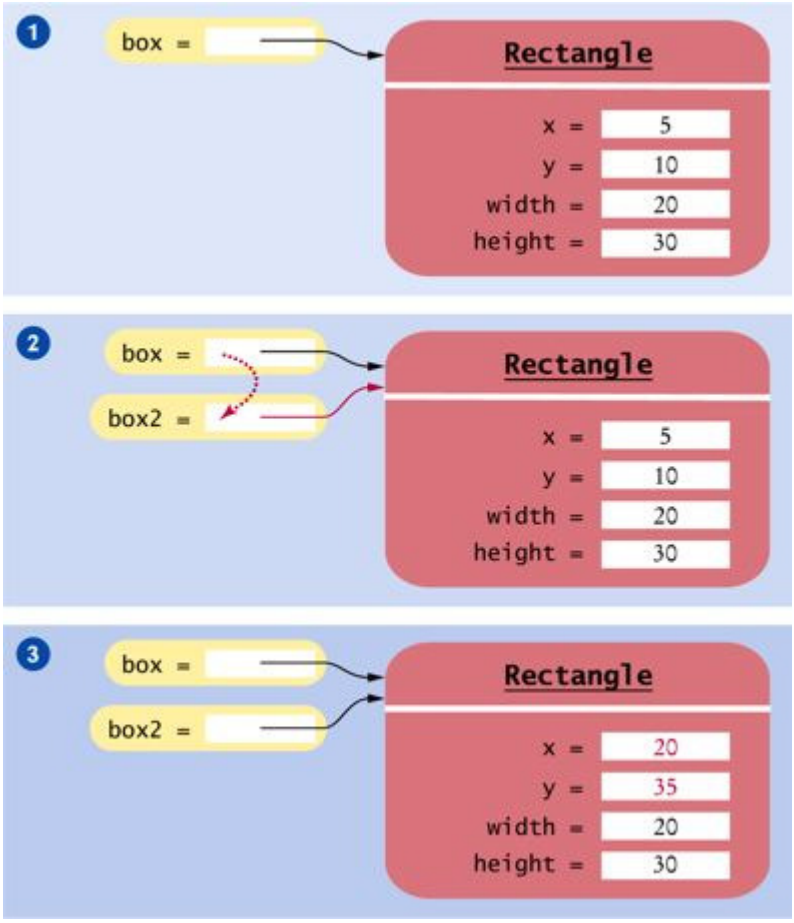
Figure 19



Copying Numbers

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

Figure 20



Copying Object References

55

56

There is a reason for the difference between numbers and objects. In the computer, each number requires a small amount of memory. But objects can be very large. It is far more efficient to manipulate only the memory location.

Frankly speaking, most programmers don't 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 “the object reference stored in box”. The difference between objects and object

references only becomes apparent when you have multiple variables that refer to the same object.

Self Check

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

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

Random Fact 2.1: Mainframes—When Dinosaurs Ruled the Earth

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.

The so-called mainframe computers of the 1950s, 1960s, and 1970s were huge. They filled rooms, which had to be climate-controlled to protect the delicate equipment (see A Mainframe Computer). Today, because of miniaturization technology, even mainframes are getting smaller, but they are still very expensive. (At the time of this writing, the cost for a typical mainframe is several million dollars.)

These huge and expensive systems were an immediate success when they first appeared, because they replaced many roomfuls of even more expensive employees, who had previously performed the tasks by hand. Few of these computers do any exciting computations. They keep mundane information, such as billing records or airline reservations; they just keep lots of them.

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 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. In the 1960s, IBM's competitors, the so-called “Seven Dwarfs”—GE, RCA, Univac, Honeywell, Burroughs, Control Data, and NCR—fell on hard times. Some went out of the computer business altogether, while others tried unsuccessfully to combine their strengths by merging their computer operations. It was generally predicted that they would eventually all fail. It was in this atmosphere that the U.S. government brought an antitrust suit against IBM in 1969. The suit went to trial in 1975 and dragged on until 1982, when the Reagan Administration abandoned it, declaring it “without merit”.

A Mainframe Computer

Of course, by then the computing landscape had changed completely. Just as the dinosaurs gave way to smaller, nimbler creatures, three new waves of computers had appeared: the minicomputers, workstations, and microcomputers, all engineered by new companies, not the Seven Dwarfs. Today, the importance of

mainframes in the marketplace has diminished, and IBM, while still a large and resourceful company, no longer dominates the computer market.

Mainframes are still in use today for two reasons. They still excel at handling large data volumes. More importantly, the programs that control the business data have been refined over the last 30 or more years, fixing one problem at a time. Moving these programs to less expensive computers, with different languages and operating systems, is difficult and error-prone. In the 1990s, Sun Microsystems, a leading manufacturer of workstations and servers—and the inventor of Java—was eager to prove that its mainframe system could be "downsized" and replaced by its own equipment. Sun eventually succeeded, but it took over five years—far longer than it expected.

57

58

2.11 Graphical Applications and Frame Windows

This is the first of several sections that teach you how to write *graphical applications*: applications that display drawings inside a window. Graphical applications look more attractive than the console applications that show plain text in a console window.

The material in this section, as well as the sections labeled "Graphics Track" in other chapters, are entirely optional. Feel free to skip them if you are not interested in drawing graphics.

A graphical application shows information inside a frame window: a window with a title bar, as shown in Figure 21. In this section, you will learn how to display a frame window. In Section 3.9, you will learn how to create a drawing inside the frame.

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

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.

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

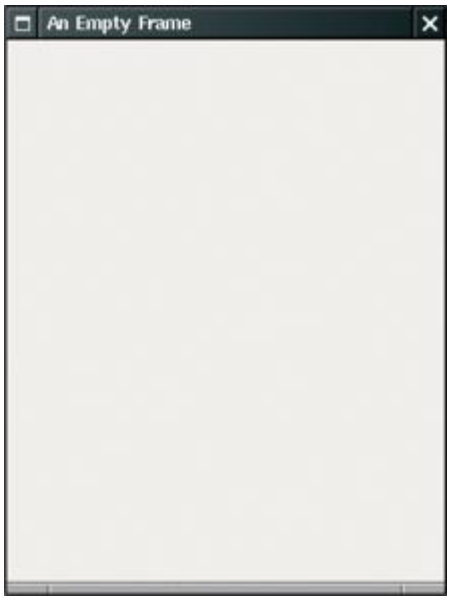
```
frame.setTitle("An Empty Frame");
```

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

4. Set the “default close operation”:

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

Figure 21



A Frame Window

58

59

When the user closes the frame, the program automatically exits. Don't omit this step. If you do, the program continues 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 21.

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.

We will go into much greater detail about Swing programming in Chapters 3, 9, 10, and 18. For now, consider this program to be the essential plumbing that is required to show a frame.

ch02/emptyframe/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
9           frame.setSize(300, 400);
10          frame.setTitle("An Empty Frame");
11          frame.setDefaultCloseOperation(JFrame.EXIT_(
12
13          frame.setVisible(true);
14       }
15 }
```

SELF CHECK

27. How do you display a square frame with a title bar that reads "Hello, World!"?

28. How can a program display two frames at once?

2.12 Drawing on a Component

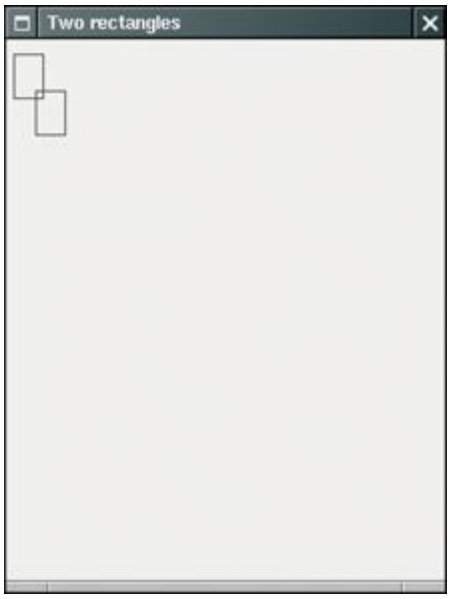
This section continues the optional graphics track. You will learn how to make shapes appear inside a frame window. The first drawing will be exceedingly modest: just two

rectangles (see Figure 22). 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. Whenever you want to show anything inside a frame, be it a button or a drawing, you have to construct a *component* object and add it to the frame. In the Swing toolkit, the `JComponent` class represents a blank component.

59

60

Figure 22



Drawing Rectangles

Since 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 define a new class that extends the `JComponent` class. You will learn about the process of extending classes in Chapter 10. For now, simply use the following code as a template.

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

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

```
            {
                  Drawing instructions go here
            }
      }
```

The `extends` keyword indicates that our component class, `RectangleComponent`, inherits the methods of `JComponent`. However, the `RectangleComponent` is different from the plain `JComponent` in one respect: The `paintComponent` method will contain instructions to draw the rectangles.

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

When the window 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.

The `paintComponent` method receives an object of type `Graphics`. The `Graphics` object stores the graphics state—the current color, font, and so on, that are used for drawing operations.

> The `Graphics` class lets you manipulate the graphics state (such as the current color).

However, the `Graphics` class is primitive. When programmers clamored for a more object-oriented approach for 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 a parameter of type `Graphics2D`. Programs with simple graphics do not need this feature. Because we want to use the more sophisticated methods to draw two-dimensional graphics objects, we need to recover the `Graphics2D`. This is accomplished by using a *cast*:

60
61

> The `Graphics2D` class has methods to draw shape objects.

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

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

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);
            . . .
      }
}
```

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.

ch02/rectangles/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          // RecoverGraphics2D
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    }
```

61

62

In order to see the drawing, one task remains. You need to display the frame into which you added a component object. Follow these steps:

1. Construct a frame as described in the preceding section.
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, as described in the preceding section.

The following listing shows the complete process.

ch02/rectangles/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          frame.setSize(300, 400);
9          frame.setTitle("Two rectangles");
10         frame.setDefaultCloseOperation(JFrame.EXI
11
12         RectangleComponent component = new
RectangleComponent();
13         frame.add(component);
14
15         frame.setVisible(true);
16      }
17   }
```

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

62

63

SELF CHECK

29. How do you modify the program to draw two squares?
30. How do you modify the program to draw one rectangle and one square?
31. What happens if you call `g.draw(box)` instead of `g2.draw(box)`?

ADVANCED TOPIC 2.2: Applets

In the preceding section, you learned how to write a program that displays graphical shapes. Some people prefer to use applets for learning about graphics programming. Applets have two advantages. They don't need separate component and viewer classes; you only implement a single class. And, more importantly, applets run inside a web browser, allowing you to place your creations on a web page for all the world to admire.

Applets are programs that run inside a web browser.

To implement an applet, use this code outline:

```
public class MyApplet extends JApplet
{
      public void paint(Graphics g)
      {
            // Recover Graphics2D
            Graphics2D g2 = (Graphics2D) g;
            // Drawing instructions go here
            . . .
      }
}
```

This is almost the same outline as for a component, with two minor differences:

1. You extend `JApplet`, not `JComponent`.
2. You place the drawing code inside the `paint` method, not inside `paintComponent`.

The following applet draws two rectangles:

ch02/applet/RectangleApplet.java

```
1     import java.awt.Graphics;
2     import java.awt.Graphics2D;
3     import java.awt.Rectangle;
4     import javax.swing.JApplet;
5
```

```
 6    /**
 7          An applet that draws two rectangles.
 8    */
 9    public class RectangleApplet extends
JApplet
10    {
11          public void paint(Graphics g)
12          {                                                    63
13                // Prepare for extended graphics              64
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    }
```

To run this applet, you need an HTML file with an `applet` tag. HTML, the hypertext markup language, is the language used to describe web pages. (See Appendix H for more information on HTML.) Here is the simplest possible file to display the rectangle applet:

To run an applet, you need an HTML file with the applet tag.

ch02/applet/RectangleApplet.html

```
1  <applet code="RectangleApplet.class"
width="300" height="400">
2  </applet>
```

If you know HTML, you can proudly explain your creation, by adding text and more HTML tags:

ch02/applet/RectangleAppletExplained.html

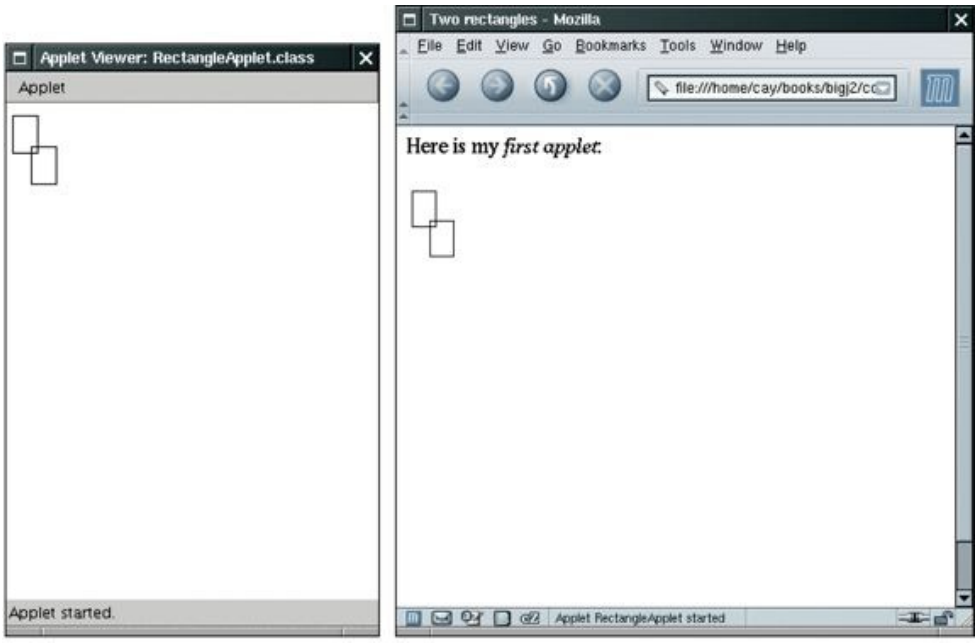
```
 1  <html>
 2        <head>
 3           <title>Two rectangles</title>
 4        </head>
 5        <body>
 6           <p>Here is my <i>first
applet</i>:</p>
 7           <applet code="RectangleApplet.class"
width="300" height="400">
 8           </applet>
 9        </body>
10  </html>
```

An HTML file can have multiple applets. Simply add a separate `applet` tag for each applet.

You can give the HTML file any name you like. It is easiest to give the HTML file the same name as the applet. But some development environments already generate an HTML file with the same name as your project to hold your project notes; then you must give the HTML file containing your applet a different name.

To run the applet, you have two choices. You can use the applet viewer, a program that is included with the Java Software Development Kit from Sun Microsystems. You simply start the applet viewer, giving it the name of the HTML file that contains your applets:

```
appletviewer RectangleApplet.html
```



An Applet in the Applet Viewer An Applet in a Web Browser

The applet viewer only shows the applet, not the HTML text (see An Applet in the Applet Viewer).

You view applets with the applet viewer or a Java-enabled browser.

You can also show the applet inside any Java 2–enabled web browser, such as Netscape or Mozilla. (If you use Internet Explorer, you probably need to configure it. By default, Microsoft supplies either an outdated version of Java or no Java at all. Go to the web site [4] and install the Java plugin.) An Applet in a Web Browser shows the applet running in a browser. As you can see, both the text and the applet are displayed.

2.13 Ellipses, Lines, Text, and Color

In Section 2.12 you learned how to write a program that draws rectangles. In this section you will learn how to draw other shapes: ellipses and lines. With these graphical elements, you can draw quite a few interesting pictures.

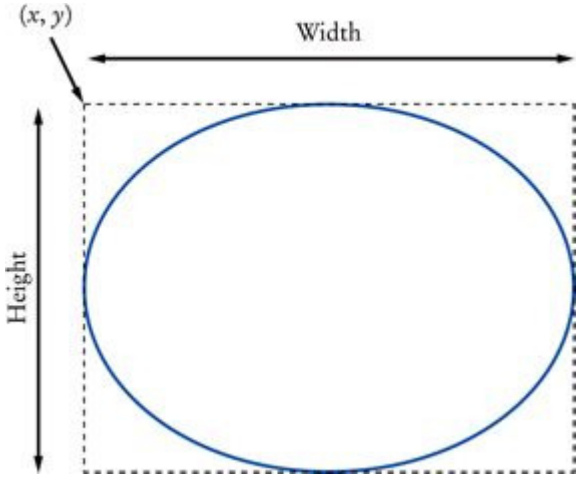
To draw an ellipse, you specify its bounding box (see Figure 23) in the same way that you would specify a rectangle, namely by the *x*- and *y*-coordinates of the top-left corner and the width and height of the box.

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:

65

66

Figure 23



An Ellipse and Its Bounding Box

```
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 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:

Ellipse2D.Double and Line2D.Double are classes that describe graphical shapes.

```
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);
```

Figure 24

Basepoint and Baseline

66

67

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

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. Simply 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.

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 24). For example,

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

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

2.13.1 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 predefined in the `Color` class. Table 1 shows those predefined colors and their RGB values. For example, `Color.PINK` has been predefined 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 rectangle 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.

67

68

Table 1 Predefined Colors and their RGB Values

Color	RGB Value
Color.BLACK	0, 0, 0
Color.BLUE	0, 0, 255
Color.CYAN	0, 255, 255
Color.GRAY	128, 128, 128
Color.DARKGRAY	64, 64, 64
Color.LIGHTGRAY	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

The following program puts all these shapes to work, creating a simple drawing (see Figure 25).

68

69

ch02/faceviewer/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.JPanel;
8  import javax.swing.JComponent;
9
10 /**
11       A component that draws an alien face.
12 */
13 public class FaceComponent extends JComponent
14 {
```

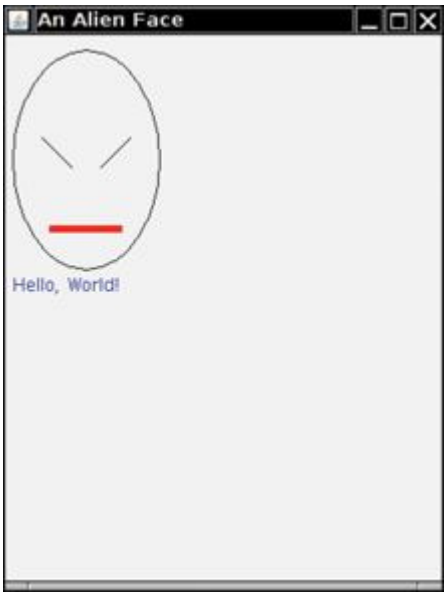
```
15          public void paintComponent(Graphics g)
16          {
17             // Recover Graphics2D
18             Graphics2D g2 = (Graphics2D) g;
19
20             // Draw the head
21             Ellipse2D.Double head = new
Ellipse2D.Double(5, 10, 100, 150);
22             g2.draw(head);
23
24             // Draw the eyes
25             Line2D.Double eye1 = new
Line2D.Double(25, 70, 45, 90);
26             g2.draw(eye1);
27
28             Line2D.Double eye2 = new
Line2D.Double(85, 70, 65, 90);
29             g2.draw(eye2);
30
31             // Draw the mouth
32             Rectangle mouth = new Rectangle(30,
130, 50, 5);
33             g2.setColor(Color.RED);
34             g2.fill(mouth);
35
36             // Draw the greeting
37             g2.setColor(Color.BLUE);
38             g2.drawString("Hello, World!", 5,
175);
39          }
40    }
```

ch02/faceviewer/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(300, 400);
9              frame.setTitle("An Alien Face");
```

```
10                frame.setDefaultCloseOperation(JFrame.
11
12                FaceComponent component = new
FaceComponent();
13                frame.add(component);
14
15                frame.setVisible(true);
16             }
17     }
```


Figure 25



An Alien Face

SELF CHECK

32. Give instructions to draw a circle with center (100, 100) and radius 25.
33. Give instructions to draw a letter "V" by drawing two line segments.
34. Give instructions to draw a string consisting of the letter "V".
35. What are the RGB color values of `Color.BLUE`?
36. How do you draw a yellow square on a red background?

RANDOM FACT 2.2: The Evolution of the Internet

In 1962, J.C.R. Licklider was head of the first computer research program at DARPA, the Defense Advanced Research Projects Agency. He wrote a series of papers describing a "galactic network" through which computer users could access data and programs from other sites. This was well before computer networks were invented. By 1969, four computers—three in California and one in Utah—were connected to the ARPANET, the precursor of the Internet. The network grew quickly, linking computers at many universities and research organizations. It was originally thought that most network users wanted to run programs on remote computers. Using remote execution, a researcher at one institution would be able to access an underutilized computer at a different site. It quickly became apparent that remote execution was not what the network was actually used for. Instead, the "killer application" was electronic mail: the transfer of messages between computer users at different locations.

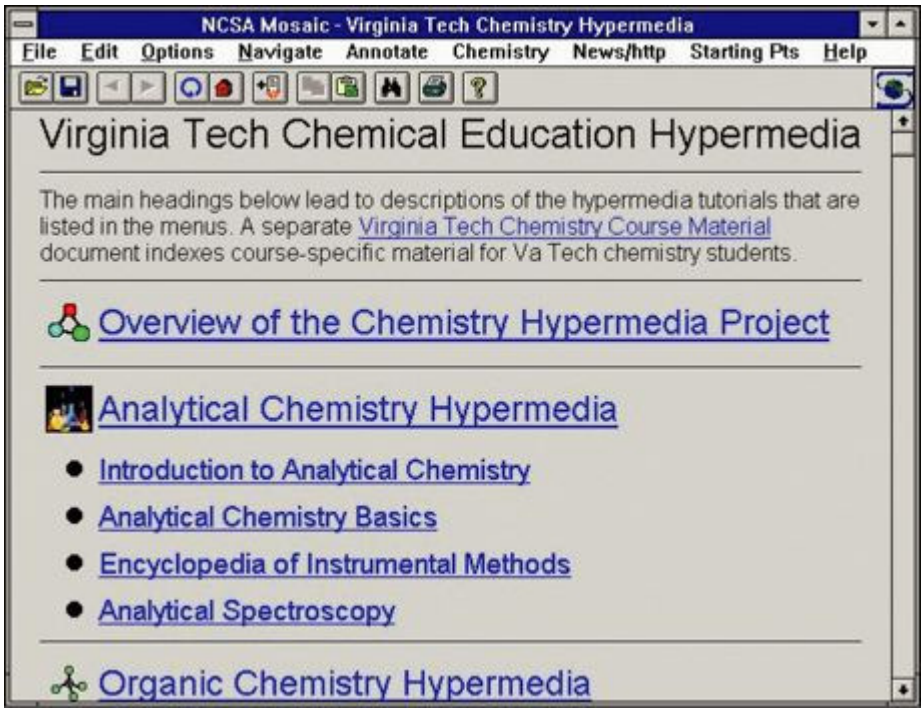
In 1972, Bob Kahn proposed to extend ARPANET into the *Internet*: a collection of interoperable networks. All networks on the Internet share common *protocols* for data transmission. Kahn and Vinton Cerf developed protocols, now called TCP/IP (Transmission Control Protocol/Internet Protocol). On January 1, 1983, all hosts on the Internet simultaneously switched to the TCP/IP protocol (which is used to this day).

Over time, researchers, computer scientists, and hobbyists published increasing amounts of information on the Internet. For example, the GNU (GNU's Not UNIX) project is producing a free set of high-quality operating system utilities and program development tools [5]. Project Gutenberg makes available the text of important classical books, whose copyright has expired, in computer-readable form [6]. In 1989, Tim Berners-Lee started work on hyperlinked documents, allowing users to browse by following links to related documents. This infrastructure is now known as the World Wide Web (WWW).

The first interfaces to retrieve this information were, by today's standards, unbelievably clumsy and hard to use. In March 1993, WWW traffic was 0.1% of all Internet traffic. All that changed when Marc Andreesen, then a graduate student working for NCSA (the National Center for Supercomputing

70
71

Applications), released Mosaic. Mosaic displayed web pages in graphical form, using images, fonts, and colors (see The NCSA Mosaic Browser figure). Andreesen went on to fame and fortune at Netscape, and Microsoft licensed the Mosaic code to create Internet Explorer. By 1996, WWW traffic accounted for more than half of the data transported on the Internet.



The NCSA Mosaic Browser

CHAPTER SUMMARY

1. In Java, every value has a type.
2. You use variables to store values that you want to use at a later time.
3. Identifiers for variables, methods, and classes are composed of letters, digits, and underscore characters.
4. By convention, variable names should start with a lowercase letter.

5. Use the assignment operator (=) to change the value of a variable.
6. All variables must be initialized before you access them.
7. Objects are entities in your program that you manipulate by calling methods.

8. A method is a sequence of instructions that accesses the data of an object.
9. A class defines the methods that you can apply to its objects.
10. The public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.
11. A parameter is an input to a method.
12. The implicit parameter of a method call is the object on which the method is invoked.
13. The return value of a method is a result that the method has computed for use by the code that called it.
14. A method name is overloaded if a class has more than one method with the same name (but different parameter types).
15. The `double` type denotes floating-point numbers that can have fractional parts.
16. In Java, numbers are not objects and number types are not classes.
17. Numbers can be combined by arithmetic operators such as `+`, `-`, and `*`.
18. Use the `new` operator, followed by a class name and parameters, to construct new objects.
19. An accessor method does not change the state of its implicit parameter. A mutator method changes the state.
20. Determining the expected result in advance is an important part of testing.
21. Java classes are grouped into packages. Use the `import` statement to use classes that are defined in other packages.

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

23. An object reference describes the location of an object.

24. Multiple object variables can contain references to the same object.

25. Number variables store numbers. Object variables store references.

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

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

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

29. The `Graphics` class lets you manipulate the graphics state (such as the current color).

30. The `Graphics2D` class has methods to draw shape objects.

72

73

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

32. Applets are programs that run inside a web browser.

33. To run an applet, you need an HTML file with the applet tag.

34. You view applets with the applet viewer or a Java-enabled browser.

35. `Ellipse2D.Double` and `Line2D.Double` are classes that describe graphical shapes.

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

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

FURTHER READING

1. http://www.bluej.org The BlueJ development environment.
2. http://drjava.sourceforge.net The Dr. Java development environment.
3. http://java.sun.com/javase/6/docs/api/index.html The documentation of the Java API.
4. http://java.com The consumer-oriented web site for Java technology. Download the Java plugin from this site.
5. http://www.gnu.org The web site of the GNU project.
6. http://www.gutenberg.org The web site of Project Gutenberg, offering the text of classical books.

CLASSES, OBJECTS, AND METHODS 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
    translate
    getX
    getY
    getHeight
```

```
    getWidth
java.lang.String
    length
    replace
    toLowerCase
    toUpperCase
javax.swing.JComponent
    paintComponent
javax.swing.JFrame
    setDefaultCloseOperation
```

73

74

REVIEW EXERCISES

★ **Exercise R2.1.** Explain the difference between an object and an object reference.

★ **Exercise R2.2.** Explain the difference between an object and an object variable.

★ **Exercise R2.3.** Explain the difference between an object and a class.

★★ **Exercise R2.4.** Give the Java code for constructing an *object* of class `Rectangle`, and for declaring an *object variable* of class `Rectangle`.

★★ **Exercise R2.5.** Explain the difference between the = symbol in Java and in mathematics.

★★★ **Exercise R2.6.** Uninitialized variables can be a serious problem. Should you always initialize every `int` or `double` variable with zero? Explain the advantages and disadvantages of such a strategy.

★★ **Exercise R2.7.** Give Java code to construct the following objects:

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

b. A string `"Hello, Dave!"`

Create objects, not object variables.

★★ **Exercise R2.8.** Repeat Exercise R2.7, but now define object variables that are initialized with the required objects.

★★ **Exercise R2.9.** 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!");
```

★ **Exercise R2.10.** Name two accessor methods and two mutator methods of the `Rectangle` class.

★★ **Exercise R2.11.** 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);
```

If you are not sure, write a small test program or use BlueJ.

★ **Exercise R2.12.** Find an overloaded method of the `String` class.

★ **Exercise R2.13.** Find an overloaded method of the `Rectangle` class.

★G **Exercise R2.14.** What is the difference between a console application and a graphical application?

★★G **Exercise R2.15.** Who calls the `paintComponent` method of a component? When does the call to the `paintComponent` method occur?

★★G **Exercise R2.16.** Why does the parameter of the `paintComponent` method have type `Graphics` and not `Graphics2D`?

★★G **Exercise R2.17.** What is the purpose of a graphics context?

★★G **Exercise R2.18.** Why are separate viewer and component classes used for graphical programs?

★G **Exercise R2.19.** How do you specify a text color?

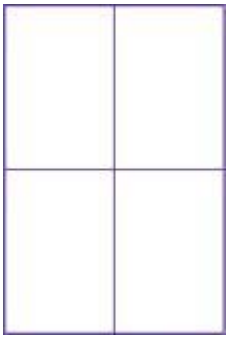
- Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★T **Exercise P2.1.** 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.

★T **Exercise P2.2.** 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.

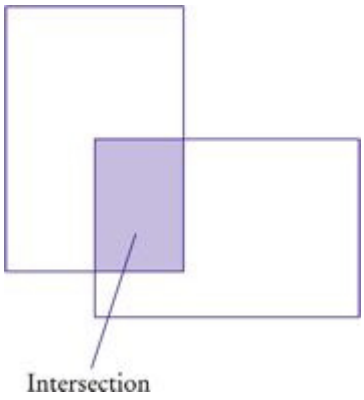
★★ **Exercise P2.3.** Write a program called `FourRectanglePrinter` that constructs a `Rectangle` object, prints its location by calling `System.out.println(box)`, and then translates and prints it three more times, so that, if the rectangles were drawn, they would form one large rectangle:



75

76

★★★ **Exercise P2.4.** The `intersection` method computes the *intersection* of two rectangles—that is, the rectangle that is formed by two overlapping rectangles:



You call this method as follows:

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

Write a program `IntersectionPrinter` that constructs two rectangle objects, prints them, 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.

★★ **Exercise P2.5.** In the Java library, a color is specified by its red, green, and blue components between 0 and 255. 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 and print the red, green, and blue values of the resulting color. (You won't actually see the color—see Section 2.13 on how to display the color.)

★★ **Exercise P2.6.** Repeat Exercise P2.5, but apply the `darker` method twice to the predefined object `Color.RED`. Call your class `DarkerDemo`.

★★ **Exercise P2.7.** 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.

★★★ **Exercise P2.8.** 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. (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.

76

77

★★T **Exercise P2.9.** 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.

★★★ **Exercise P2.10.** 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!"`

★★G **Exercise P2.11.** Write a graphics program that draws your name in red, contained inside a blue rectangle. Provide a class `NameViewer` and a class `NameComponent`.

★★G **Exercise P2.12.** Write a graphics program that draws 12 strings, one each for the 12 standard colors, besides `Color.WHITE`, each in its own color. Provide a class `Color-NameViewer` and a class `ColorNameComponent`.

★★G **Exercise P2.13.** 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`.

★★★G **Exercise P2.14.** 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.

★★G **Exercise P2.15.** Write a program to plot the following face.

Provide a class `FaceViewer` and a class `FaceComponent`.

- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 2.1.** 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 constants `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 following information:

- 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.

★★★G **Project 2.2.** Run the following program:

```
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class FrameTester
{
     public static void main(String[] args)
     {
          JFrame frame = new JFrame();
          frame.setSize(200, 200);
          JTextField text = new JTextField
("Hello, World!");
          text.setBackground(Color.PINK);
          frame.add(text);
          frame.setDefaultCloseOperation(JFrame.EXIT_
          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 P2.5)

ANSWERS TO SELF-CHECK QUESTIONS

1. `int` and `String`
2. Only the first two are legal identifiers.
3. `String myName = "John Q. Public"`
4. No, the left-hand side of the = operator must be a variable.
5. `greeting = "Hello, Nina!";`

   Note that

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

   is not the right answer—that statement defines a new variable.
6. `river.length()` or `"Mississippi".length()`
7. `System.out.println(greeting.toUpperCase());`
8. It is not legal. The variable `river` has type `String`. The `println` method is not a method of the `String` class.
9. The implicit parameter is `river`. There is no explicit parameter. The return value is 11.
10. `"Mississi"`
11. 12
12. As `public String toUpperCase()`, with no explicit parameter and return type `String`.
13. `double`
14. An `int` is not an object, and you cannot call a method on it.
15. `(x + y) * 0.5`
16. `new Rectangle(90, 90, 20, 20)`

17. 0

18. An accessor—it doesn't modify the original string but returns a new string with uppercase letters.

19. `box.translate(-5, -10)`, provided the method is called immediately after storing the new rectangle into `box`.

20. `x: 30, y: 25`

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

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

23. `toLowerCase`

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

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

26. 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.

27. Modify the `EmptyFrameViewer` program as follows:

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

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

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

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

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

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

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

33.
```
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);
```

34.
```
g2.drawString("V", 0, 30);
```

35.
```
0, 0, 255
```

36. 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 simple methods
- To understand the purpose and use of constructors
- To understand how to access instance fields and local variables
- To appreciate the importance of documentation comments
- **G** To implement classes for drawing graphical shapes

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. You then need to implement the methods. This step requires that you find a data representation for the objects, and supply the instructions for each method. You then provide a tester to validate that your class works correctly. You also document your efforts so that other programmers can understand and use your creation.

3.1 Levels of Abstraction

3.1.1 Black Boxes

When you lift the hood of a car, you will find a bewildering collection of mechanical components. You will probably recognize the motor and the tank for the wind-shield washer fluid. Your car mechanic will be able to identify many other components, such as the transmission and the electronic control module—the device that controls the timing of the spark plugs and the flow of gasoline into the motor. But ask your mechanic what is inside the electronic control module, and you will likely get a shrug.

It is a *black box*, something that magically does its thing. A car mechanic would never open the box—it contains electronic parts that can only be serviced at the factory. Of course, the device may have a color other than black, and it may not even be box-shaped. But 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 interaction with the outside world is well-defined. For example, the car mechanic can test that the engine control module sends the right firing signals to the spark plugs.

82

83

Why do car manufacturers put black boxes into cars? The black box greatly simplifies the work of the car mechanic, leading to lower repair costs. If the box fails, it is simply replaced with a new one. Before engine control modules were invented, gasoline flow into the engine was regulated by a mechanical device called a carburetor, a notoriously fussy mess of springs and latches that was expensive to adjust and repair.

Of course, for many drivers, the *entire car* is a "black box". Most drivers know nothing about its internal workings and never want to open the hood in the first place. The car has pedals, buttons, and a gas tank door. If you give it the right inputs, it does its thing, transporting you from here to there.

And for the engine control module manufacturer, the transistors and capacitors that go inside are black boxes, magically produced by an electronics component manufacturer.

In technical terms, a black box provides *encapsulation*, the hiding of unimportant details. Encapsulation is very important for human problem solving. A car mechanic is more efficient when the only decision is to test the electronic control module and to replace it when it fails, without having to think about the sensors and transistors inside. A driver is more efficient when the only worry is putting gas in the tank, not thinking about the motor or electronic control module inside.

However, there is another aspect to encapsulation. Somebody had to come up with the right *concept* for each particular black box. Why do the car parts manufacturers build electronic control modules and not another device? Why do the transportation device manufacturers build cars and not personal helicopters?

Concepts are discovered through the process of *abstraction*, taking away inessential features, until only the essence of the concept remains. For example, “car” is an abstraction, describing devices that transport small groups of people, traveling on the ground, and consuming gasoline. Is that the right abstraction? Or is a vehicle with an electric engine a “car”? We won't answer that question and instead move on to the significance of encapsulation and abstraction in computer science.

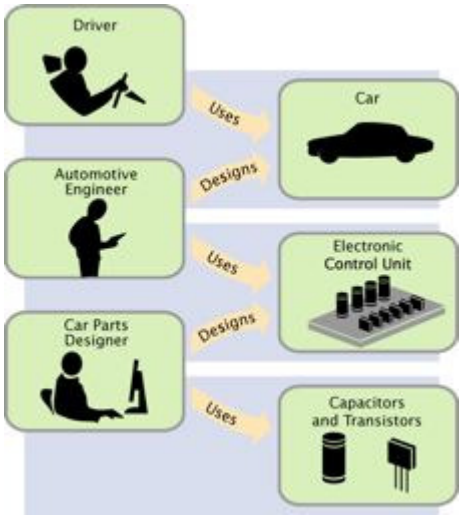
3.1.2 Object-Oriented Design

In old times, computer programs manipulated *primitive types* such as numbers and characters. As programs became more complex, they manipulated more and more of these primitive quantities, until programmers could no longer keep up. It was just too confusing to keep all that detail in one's head. As a result, programmers gave wrong instructions to their computers, and the computers faithfully executed them, yielding wrong answers.

Of course, the answer to this problem was obvious. Software developers soon learned to manage complexity. They encapsulated routine computations, forming software “black boxes” that can be put to work without worrying about the internals. They used the process of abstraction to invent data types that are at a higher level than numbers and characters.

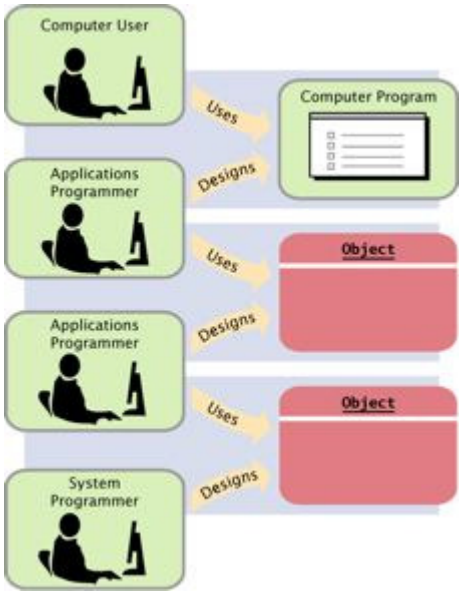
At the time that this book is written, the most common approach for structuring computer programming is the *object-oriented* approach. The black boxes from which a program is manufactured are called objects. An object has an internal structure—perhaps just some numbers, perhaps other objects—and a well-defined behavior. Of course, the internal structure is hidden from the programmer who uses it. That programmer only learns about the object's behavior and then puts it to work in order to achieve a higher-level goal.

Figure 1



Levels of Abstraction in Automotive Design

Figure 2



Levels of Abstraction in Software Design

Who designs these objects? Other programmers! What do they contain? Other objects! This is where things get confusing for beginning students. In real life, the users of black boxes are quite different from their designers, and it is easy to understand the levels of abstraction (see Figure 1). With computer programs, there are also levels of abstraction (see Figure 2), but they are not as intuitive to the uninitiated. To make matters potentially more confusing, you will often need to switch roles, being the designer of objects in the morning and the user of the same objects in the afternoon. In that regard, you will be like the builders of the first automobiles, who singlehandedly produced steering wheels and axles and then assembled their own creations into a car.

There is another challenging aspect of designing objects. Software is infinitely more flexible than hardware because it is unconstrained from physical limitations. Designers of electronic parts can exploit a limited number of physical effects to create transistors, capacitors, and the like. Transportation device manufacturers can't easily produce personal helicopters because of a whole host of physical limitations, such as fuel consumption and safety. But in software, anything goes. With few constraints from the outside world, you can design good and bad abstractions with equal facility. Understanding what makes good design is an important part of the education of a software engineer.

84

85

3.1.3 Crawl, Walk, Run

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, your role was analogous to the automotive engineer who learns how to use an engine control module, and how to take advantage of its behavior in order to build a car.

In this chapter, you will move on to implementing classes. A design will be handed to you that describes the behavior of the objects of a class. You will learn the necessary Java programming techniques that enable your objects to carry out the desired behavior. In these sections, your role is analogous to the car parts manufacturer who puts together an engine control module from transistors, capacitors, and other electronic parts.

In Chapters 8 and 12, you will learn more about designing your own classes. You will learn rules of good design, and how to discover the appropriate behavior of

objects. In those chapters, your job is analogous to the car parts engineer who specifies how an engine control module should function.

SELF CHECK

1. 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`?
2. 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?

3.2 Specifying the Public Interface of a Class

In this section, we will discuss the process of specifying the behavior 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 understand the design of a `BankAccount` class so that you can implement it, which in turn allows other programmers on the team to use it.

You need to know exactly what features of a bank account need to be implemented. Some features are essential (such as deposits), whereas others are not important (such as the gift that a customer may receive for opening a bank account). Deciding which features 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:

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

- Deposit money
- Withdraw money
- Get the current balance

In Java, operations are expressed as method calls. 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(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```

Note that 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 can print or store in a variable.

As you can see from the sample calls, the `BankAccount` class should define 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.

When you define a method, you also need to provide the method *body*, consisting of statements that are executed when the method is called.

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

You will see in Section 3.5 how to fill in the method body.

Every method definition contains the following parts:

- An *access specifier* (usually `public`)
- The *return type* (such as `void` or `double`)
- The name of the method (such as `deposit`)

- A list of the *parameters* of the method (if any), enclosed in parentheses (such as `double amount`)
- The *body* of the method: statements enclosed in braces

The access specifier controls which other methods can call this method. Most methods should be declared as `public`. That way, 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.)

A method definition contains an access specifier (usually `public`), a return type, a method name, parameters, and the method body.

The return type is the type of the output value. The `deposit` method does not return a value, whereas the `getBalance` method returns a value of type `double`.

86

87

SYNTAX 3.1 Method Definition

```
accessSpecifier returnType
methodName(parameterType parameterName, . . . )
{
   method body
}
```

Example:

```
public void deposit(double amount)
{
   . . .
}
```

Purpose:

To define the behavior of a method

Each parameter (or input) to the method has both a type and a name. For example, the `deposit` method has a single parameter named `amount` of type `double`. For each parameter, choose a name that is both a legal variable name and a good description of the purpose of the input.

Next, you need to supply constructors. We will want to construct bank accounts that initially have a zero balance, by using the default constructor:

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

What if a programmer who uses our class wants to start out with another balance? A second constructor that sets the balance to an initial value will be useful:

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

To summarize, it is specified that two constructors will be provided:

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

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`)

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

Constructors contain instructions to initialize objects. The constructor name is always the same as the class name.

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

87

88

The statements in the constructor body will set the internal data of the object that is being constructed—see Section 3.5.

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 parameters.

When defining a class, you place all constructor and method definitions inside, like this:

```
public class BankAccount
{
      // Constructors
      public BankAccount()
      {
          body—filled in later
      }
      public BankAccount(double initialBalance)
      {
          body—filled in later
      }
      // Methods
      public void deposit(double amount)
      {
          body—filled in later
      }
      public void withdraw(double amount)
      {
          body—filled in later
      }
      public double getBalance()
      {
          body—filled in later
      }
      private fields—filled in later
}
```

You will see how to supply the missing pieces in the following sections.

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. 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);
```

SYNTAX 3.2 Constructor Definition

```
accessSpecifier ClassName(parameterType
parameterName, . . . )
{
    constructor body
}
```

Example:

```
public BankAccount(double initialBalance)
{
    . . .
}
```

Purpose:

To define the behavior of a constructor

SYNTAX 3.3 Class Definition

```
accessSpecifier class ClassName
{
    constructors
    methods
    fields
}
```

Example:

```
public class BankAccount
{
    public BankAccount(double initialBalance) {. .
.}
    public void deposit(double amount) {. . .}
    . . .
}
```

Purpose:

To define a class, its public interface, and its implementation details

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

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

89
90

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 internal details. We will do so in Section 3.5. First, however, an important step remains: *documenting* the public interface. That is the topic of the next section.

SELF CHECK

3. How can you use the methods of the public interface to *empty* the `harrys-Checking` bank account?
4. 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?

3.3 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 Productivity Hint 3.1 for a description of this utility.)

A documentation comment is placed before the class or method definition 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 method parameter, you supply a line that starts with `@param`, followed by the parameter name and a short explanation. Finally, you supply a line that starts with `@return`,

describing the return value. You omit the `@param` tag for methods that have no parameters, and you omit the `@return` tag for methods whose return type is `void`.

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

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                    90
}                                                     91
/**
    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*, explaining its purpose. The comment syntax for class comments is very simple: Just place the documentation comment above the class.

```
/**
    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?" These comments do seem pretty repetitive. But you should take the time to write them, even if it feels silly.

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.

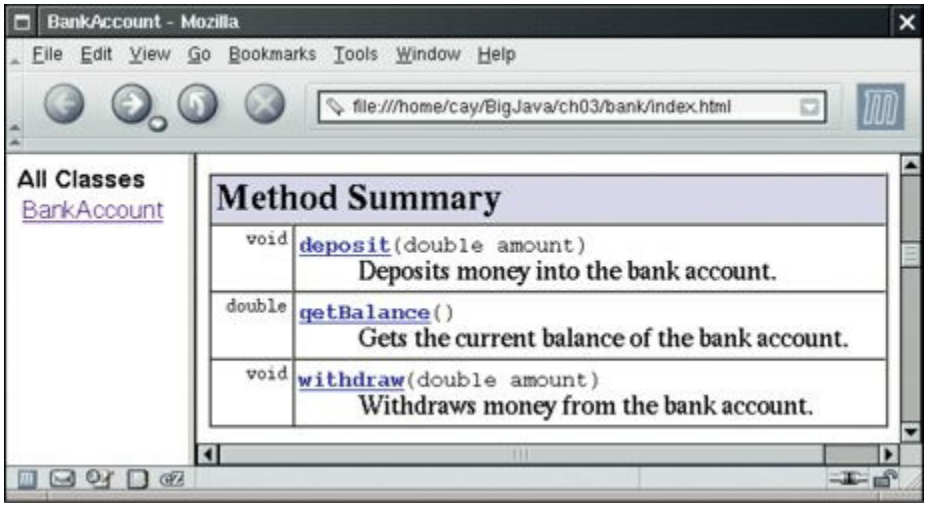
What about very simple methods? You can easily spend more time pondering whether a comment is too trivial to write than it takes to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without any comment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

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

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.

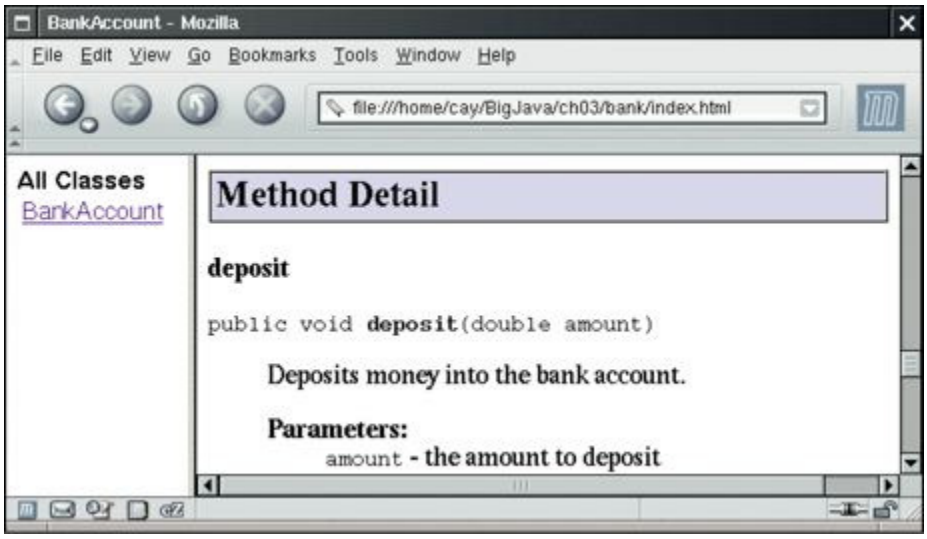
This documentation format should look familiar. The programmers who implement the Java library use `javadoc` themselves. They too document every class, every method, every parameter, and every return value, and then use `javadoc` to extract the documentation in HTML format.

Figure 3



A Method Summary Generated by `javadoc`

Figure 4



Method Detail Generated by `javadoc`

SELF CHECK

5. 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)
```

6. Why is the following documentation comment questionable?

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

92

93

PRODUCTIVITY HINT 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 command shell, 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.

3.4 Instance Fields

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

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.)

An object stores its data in *instance fields*. A *field* is a technical term for a storage location inside a block of memory. An *instance* of a class is an object of the class. Thus, an instance field is a storage location that is present in each object of the class.

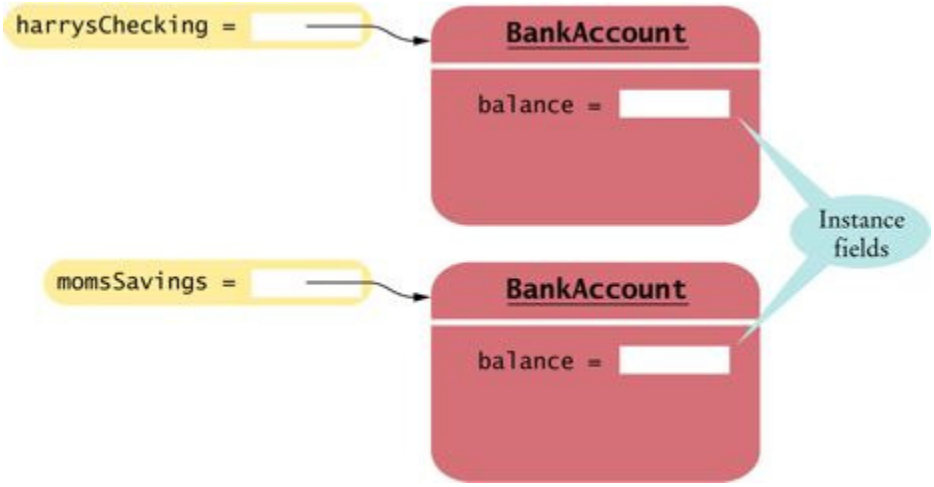
An object uses instance fields to store its state—the data that it needs to execute its methods.

The class declaration specifies the instance fields:

```
public class BankAccount
{
   . . .
   private double balance;
}
```

93

Figure 5



Instance Fields

An instance field declaration consists of the following parts:

- An *access specifier* (usually `private`)
- The *type* of the instance field (such as `double`)
- The name of the instance field (such as `balance`)

Each object of a class has its own set of instance fields. For example, if `harrysChecking` and `momsSavings` are two objects of the `BankAccount` class, then each object has its own `balance` field, called `harrysChecking.balance` and `momsSavings.balance` (see Figure 5).

> Each object of a class has its own set of instance fields.

Instance fields 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 `balance` variable can be accessed by the `deposit` method of the `BankAccount` class but not the `main` method of another class.

You should declare all instance fields as private.

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

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

In other words, if the instance fields are declared as private, then all data access must occur through the public methods. Thus, the instance fields of an object are effectively hidden from the programmer who uses a class. They are of concern only to the programmer who implements the class. The process of hiding the data and providing methods for data access is called *encapsulation*. Although it is theoretically possible in Java to leave instance fields public, that is a very uncommon practice. We will always make instance fields private in this book.

94

95

SYNTAX 3.4 Instance Field Declaration

```
accessSpecifier class ClassName
{
      . . .
      accessSpecifier fieldType fieldName;
      . . .
}
```

Example:

```
public class BankAccount
{
      . . .
      private double balance;
      . . .
```

```
    }
```

Purpose:

To define a field that is present in every object of a class

SELF CHECK

7. Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance fields?
8. What are the instance fields of the `Rectangle` class?

3.5 Implementing Constructors and Methods

Now that we have determined the instance fields, let us complete the `BankAccount` class by supplying the bodies of the constructors and methods. Each body contains a sequence of statements. We'll start with the constructors because they are very straightforward. A constructor has a simple job: to initialize the instance fields of an object.

Constructors contain instructions to initialize the instance fields 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 parameter:

```
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.

- Create a new object of type `BankAccount`.
- Call the second constructor (since a construction parameter is supplied).
- Set the parameter variable `initialBalance` to 1000.
- Set the `balance` instance field of the newly created object to `initialBalance`.
- 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.

Let's move on to implementing the `BankAccount` methods. Here is the `deposit` method:

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

To understand exactly what the method does, consider this statement:

```
harrysChecking.deposit(500);
```

This statement carries out the following steps:

- Set the parameter variable `amount` to 500.
- Fetch the `balance` field of the object whose location is stored in `harrysChecking`.
- Add the value of `amount` to `balance` and store the result in the variable `newBalance`.
- Store the value of `newBalance` in the `balance` instance field, overwriting the old value.

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

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

96

97

SYNTAX 3.5 The `return` Statement

```
return expression;
or
return;
```

Example:

```
return balance;
```

Purpose:

To specify the value that a method returns, and exit the method immediately. The return value becomes the value of the method call expression.

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

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

The `return` statement is a special statement that instructs the method to terminate and return an output to the statement that called the method. In our case, we simply return the value of the `balance` instance field. You will later see other methods that compute and return more complex expressions.

Use the `return` statement to specify the value that a method returns to its caller.

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.

ch03/account/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         /**
8            Constructs a bank account with a
zero balance.
9          */
10        public BankAccount()
11        {
12              balance = 0;
13        }
14
15      /**
16              Constructs a bank account with a
given balance.
17              @param initialBalance the initial
balance
18      */
19      public BankAccount(double initialBalance)
20      {
21            balance = initialBalance;
22      }
23
24      /**
25            Deposits money into the bank
account.
26            @param amount the amount to deposit
27      */
28      public void deposit(double amount)
29      {
30            double newBalance = balance +
amount;
31            balance = newBalance;
```

97

98

```
 32       }
 33
 34       /**
 35             Withdraws money from the bank
account.
 36             @param amount the amount to withdraw
 37       */
 38       public void withdraw(double amount)
 39       {
 40             double newBalance = balance -
amount;
 41             balance = newBalance;
 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
 53       private double balance;
 54    }
```

SELF CHECK

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

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

98

99

How To 3.1: Implementing a Class

This is the first of several "How To" sections in this book. Users of the Linux operating system have how to guides that give answers to the common questions "How do I get started?" and "What do I do next?". Similarly, the How To sections in this book give you step-by-step procedures for carrying out specific tasks.

You will often be asked to implement a class. For example, a homework assignment might ask you to implement a `CashRegister` class.

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

In the cash register example, you won't have to provide every feature of a real cash register—there are too many. The assignment should tell you *which aspects* of a cash register your class should simulate. You should have received a description, in plain English, of the operations that an object of your class should carry out, such as this one:

- Ring up the sales price for a purchased item.
- Enter the amount of payment.
- Calculate the amount of change due to the customer.

For simplicity, we are looking at a very simple cash register here. A more sophisticated model would be able to compute sales tax, daily sales totals, and so on.

Step 2 Specify the public interface.

Turn the list in Step 1 into a set of methods, with specific types for the parameters 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.enterPayment(50);
double change = register.giveChange();
```

Now we have a specific list of methods.

- `public void recordPurchase(double amount)`
- `public void enterPayment(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 fields 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 would start out with some coins and bills so that we can give exact change, but that is beyond the scope of our assignment.

Thus, we add a single constructor:

- `public CashRegister()`

99

100

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)
      {
      }

      /**
```

```
            Enters the payment received from the
    customer.
            @param amount the amount of the payment
         */
         public void enterPayment(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 fields.

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 fields and the method parameters. 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 fields to store the information that the method needs.

In the cash register example, you would want 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;
}
```

100

101

Step 5 Implement constructors and methods.

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

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

Here is the `giveChange` method. Note that 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 fields. It is common for a beginner to start out with a set of fields that cannot accurately reflect the state of an object. Don't hesitate to go back and add or modify fields.

Once you have completed the implementation, compile your class and fix any compiler errors.

Step 6 Test your class.

Write a short tester program and execute it. The tester program can 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.enterPayment(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
```

Alternatively, if you use a program that lets you test objects interactively, such as BlueJ, construct an object and apply the method calls.

101

102

3.6 Unit Testing

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.

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

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*.

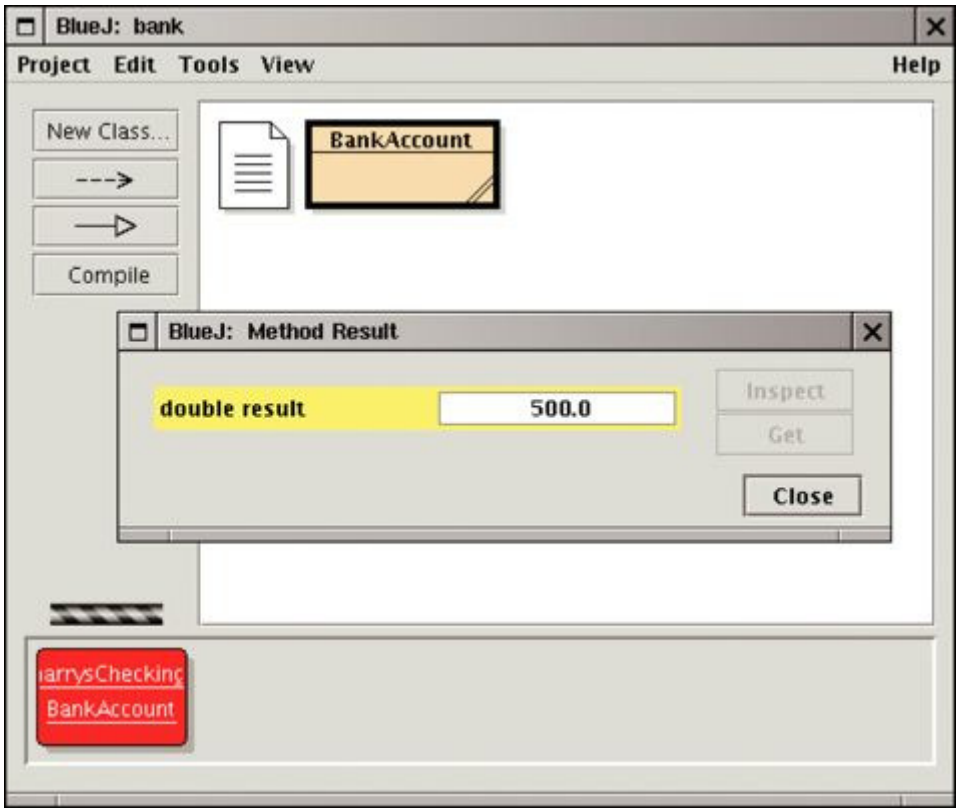
To test your class, you have two choices. Some interactive development environments have commands for constructing objects and invoking methods (see Advanced Topic 2.1). Then you can test a class simply by constructing an object, calling methods, and verifying that you get the expected return values.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

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. A tester class typically carries out the following steps:

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

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

102

103

The `MoveTester` class in Section 2.8 is a good example of a tester class. That class runs methods of the `Rectangle` class—a class in the Java library.

Here 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 $1500.

ch03/account/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 }
```

Output

```
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 `Bank-Account` class describes objects that compute bank balances. The `BankAccountTester` class runs a test that puts a `BankAccount` object through its paces.

103

104

Self Check

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

Productivity Hint 3.2: Using the Command Line Effectively

If your programming environment allows you to accomplish all routine tasks using menus and dialog boxes, you can skip this note. However, if you must invoke the editor, the compiler, the linker, and the program to test manually, then it is well worth learning about *command line editing*.

Most operating systems (including Linux, Mac OS X, UNIX, and Windows) have a *command line interface* to interact with the computer. (In Windows XP, you can get a command line window by selecting "Run …" from the Start menu and typing `cmd`.) You launch commands at a *prompt*. The command is executed, and on completion you get another prompt.

When you develop a program, you find yourself executing the same commands over and over. Wouldn't it be nice if you didn't have to type commands, such as

```
javac MyProg.java
```

more than once? Or if you could fix a mistake rather than having to retype the command in its entirety? Many command line interfaces have an option to do just that, by using the up and down arrow keys to recall old commands and the left and right arrow keys to edit lines. You can also perform *file completion*. For example, to select the file `BankAccount.java`, you only need to type the first couple of letters and then hit the "Tab" key.

The details depend on your operating system and its configuration—experiment on your own, or ask a "power user" for help.

3.7 Categories of Variables

We close this chapter with two sections of a more technical nature, examining variables and parameters in some detail.

You have seen three different categories of variables in this chapter:

1. *Instance fields* (sometimes called *instance variables*), such as the `balance` variable of the `BankAccount` class
2. *Local variables*, such as the `newBalance` variable of the `deposit` method
3. *Parameter variables*, such as the `amount` variable of the `deposit` method

104

105

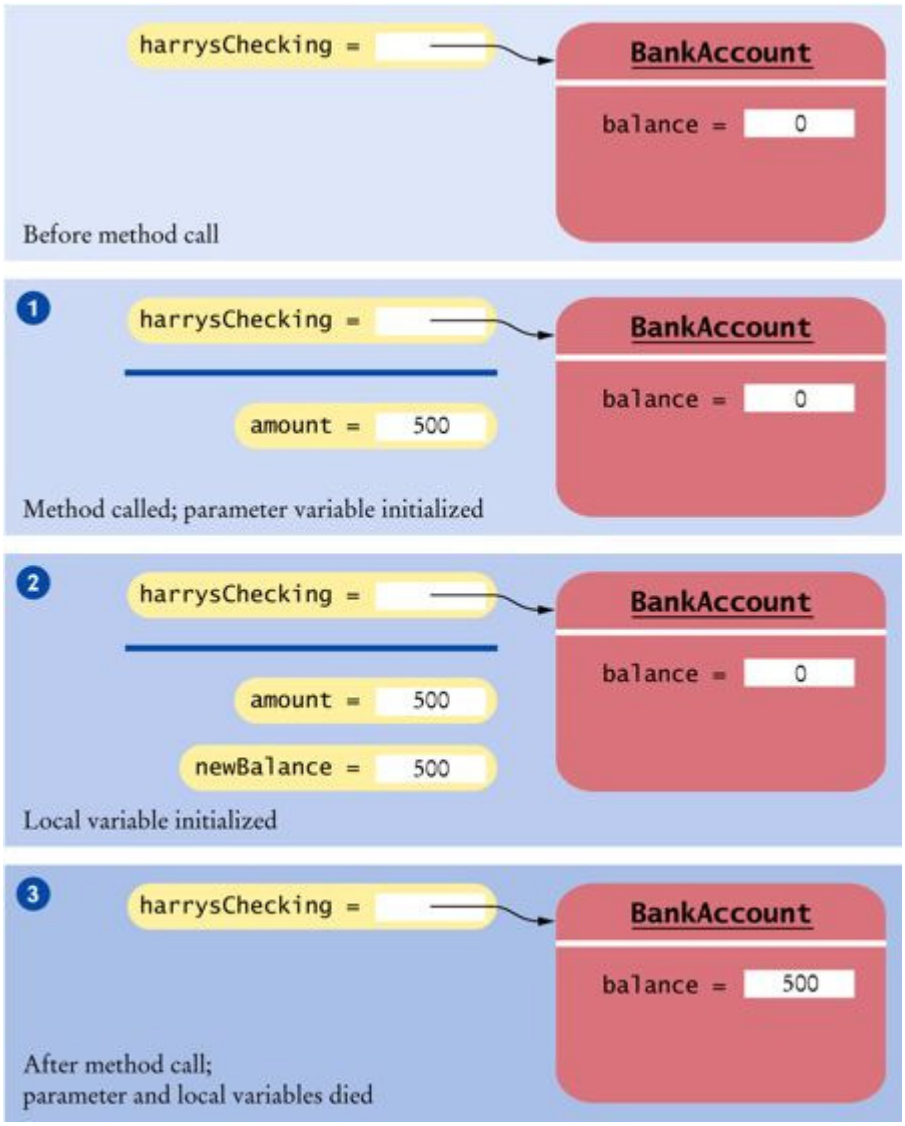
These variables are similar in one respect—they all hold values that belong to specific types. But they have a couple of important differences. The first difference is their *lifetime*.

Instance fields belong to an object. Parameter variables and local variables belong to a method—they die when the method exits.

An instance field belongs to an object. Each object has its own copy of each instance field. For example, if you have two `BankAccount` objects (say, `harrysChecking` and `momsSavings`), then each of them has its own `balance` field. When an object is constructed, its instance fields are created. The fields 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.)

Local and parameter variables belong to a method. When the method runs, these variables come to life. When the method exits, they die immediately (see Figure 7).

Figure 7



Lifetime of Variables

105

106

For example, if you call

```
harrysChecking.deposit(500);
```

then a parameter variable called `amount` is created and initialized with the parameter value, 500. When the method returns, the `amount` variable dies. The same holds for the local variable `newBalance`. When the `deposit` method reaches the line

```
double newBalance = balance + amount;
```

the variable comes to life and is initialized with the sum of the object's balance and the deposit amount. The lifetime of that variable extends to the end of the method.

However, the `deposit` method has a lasting effect. Its next line,

```
balance = newBalance;
```

sets the `balance` instance field, and that field lives beyond the end of the `deposit` method, as long as the `BankAccount` object is in use.

The second major difference between instance fields 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.

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

Parameter variables are initialized with the values that are supplied in the method call.

Instance fields are initialized with a default value if you don't explicitly set them in a constructor. Instance fields 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. Inadvertent initialization with 0 or `null` is a common cause of errors. Therefore, it is a matter of good style to initialize *every* instance field explicitly in every constructor.

Self Check

13. What do local variables and parameter variables have in common? In which essential aspect do they differ?

14. During execution of the `BankAccountTester` program in the preceding section, how many instance fields, local variables, and parameter variables were created, and what were their names?

COMMON ERROR 3.1: 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 fields. Every constructor needs to ensure that all instance fields are set to appropriate values.

106

107

If you do not initialize an instance field, 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
{
    public BankAccount() {} // No statements
    . . .
    private double balance;
    private String owner;
}
```

The `balance` is set to 0, and the `owner` field is set to a `null` reference. This is a problem—it is illegal to call methods on the `null` reference.

If you forget to initialize a *local* variable in a *method*, the compiler flags this as an error, and you must fix it before the program runs. If you make the same mistake with an *instance* field in a class, the compiler provides a default initialization, and the error becomes apparent only when the program runs.

To avoid this problem, make it a habit to initialize every instance field in every constructor.

3.8 Implicit and Explicit Method Parameters

In Section 2.4, you learned that a method has an implicit parameter—the object on which the method is invoked—and explicit parameters, which are enclosed in parentheses. In this section, we will examine these parameters in greater detail.

Have a look at a particular invocation of the `deposit` method:

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

Now look again at the code of the `deposit` method:

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

The parameter variable `amount` is set to 500 when the `deposit` method starts. But what does `balance` mean exactly? After all, our program may have multiple `BankAccount` objects, and *each of them* has its own balance.

Of course, since we deposit the money into `momsSavings`, `balance` must mean `momsSavings.balance`. In general, when you refer to an instance field inside a method, it means the instance field of the object on which the method was called.

107

108

Thus, the call to the `deposit` method depends on two values: the object to which `momsSavings` refers, and the value `500`. The `amount` parameter inside the parentheses is called an *explicit* parameter, because it is explicitly named in the method definition. However, the reference to the bank account object is not explicit in the method definition—it is called the *implicit parameter* of the method.

> The implicit parameter of a method is the object on which the method is invoked. The `this` reference denotes the implicit parameter.

If you need to, you can access the implicit parameter—the object on which the method is called—with the keyword `this`. For example, in the preceding method invocation, `this` was set to `momsSavings` and `amount` was set to 500 (see Figure 8).

Every method has one implicit parameter. You don't give the implicit parameter a name. It is always called `this`. (There is one exception to the rule that every method has an implicit parameter: `static` methods do not. We will discuss them in Chapter 8.) In contrast, methods can have any number of explicit parameters—which you can name any way you like—or no explicit parameter at all.

Next, look closely at the implementation of the `deposit` method. The statement

```
double newBalance = balance + amount;
```

actually means

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

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

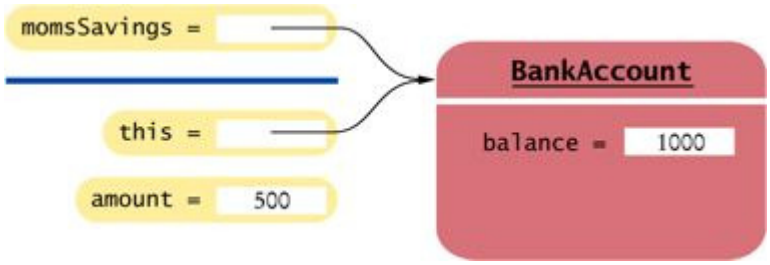
Use of an instance field name in a method denotes the instance field of the implicit parameter.

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

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

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

Figure 8



The Implicit Parameter of a Method Call

108

109

SELF CHECK

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

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

17. How many implicit and explicit parameters does the `main` method of the `BankAccount-Tester` class have, and what are they called?

COMMON ERROR 3.2: Trying to Call a Method Without an Implicit Parameter

Suppose your `main` method contains the instruction

```
withdraw(30); // Error
```

The compiler will not know which account to access to withdraw the money. You need to supply an object reference of type `BankAccount`:

```
BankAccount harrysChecking = new BankAccount();
harrysChecking.withdraw(30);
```

However, there is one situation in which it is legitimate to invoke a method without, seemingly, an implicit parameter. Consider the following modification to the `BankAccount` class. Add a method to apply the monthly account fee:

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

That means to withdraw from the same bank account object that is carrying out the `monthly-Fee` 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 always use the `this` parameter to make the method easier to read:

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

109

110

ADVANCED TOPIC 3.1: Calling One Constructor from Another

Consider the `BankAccount` class. It has two constructors: a constructor without parameters 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 constructor call can occur only as the *first line in another constructor*.

This syntax is a minor convenience. We will not use it in this book. Actually, the use of the keyword `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 this class.

RANDOM FACT 3.1: Electronic Voting Machines

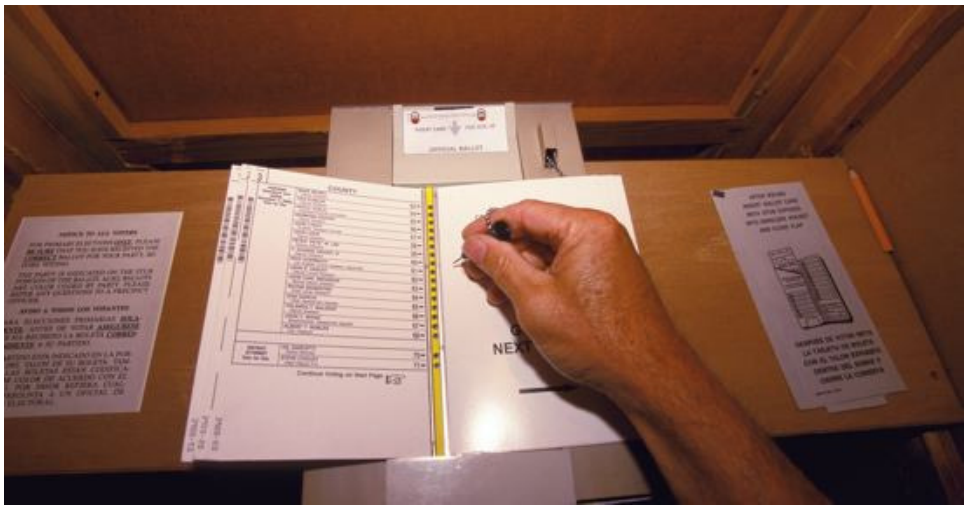
In the 2000 presidential elections in the United States, votes were tallied by a variety of machines. Some machines processed cardboard ballots into which voters punched holes to indicate their choices (see Punch Card Ballot figure). When voters were not careful, remains of paper—the now infamous "chads"—were 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.

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 an automatic bank teller machine (see Touch Screen Voting Machine figure).

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.

110
111



Punch Card Ballot

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 be complemented by a *voter verifiable audit trail*. (A good source of

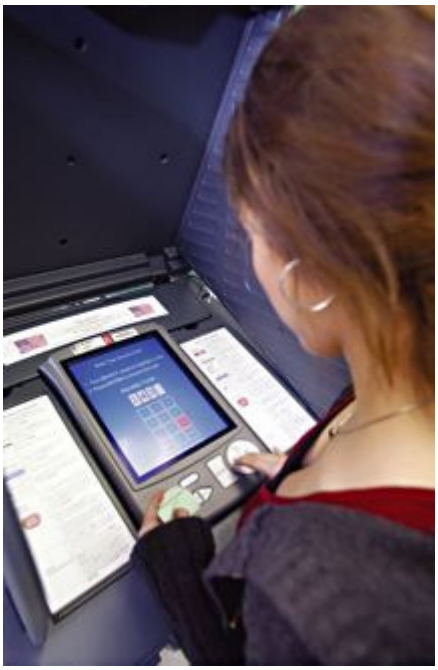
information is [1].) Typically, a voter-verifiable machine prints out the choices that are being tallied. Each voter has a chance to review the printout, and then deposits it in an old-fashioned ballot box. If there is a problem with the electronic equipment, the printouts can be counted by hand.

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 automatic 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?

At any rate, 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.

111
112



Touch Screen Voting Machine

3.9 Shape Classes

We continue the optional graphics track by discussing how to organize complex drawings in a more object-oriented fashion. Feel free to skip this section if you are not interested in graphical applications.

When you produce a drawing that is composed of complex parts, such as the one in Figure 9, it is a good idea to make a separate class for each part. Provide a `draw` method that draws the shape, and provide a constructor to set the position of the shape. For example, here is the outline of the `Car` class.

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

```
public class Car
{
```

```
      public Car(int x, int y)
      {
         // Remember position
         . . .
      }
      public void draw(Graphics2D g2)
      {
         // Drawing instructions
         . . .
      }
   }
```

112

113

Figure 9

The Car Component Draws Two Car Shapes

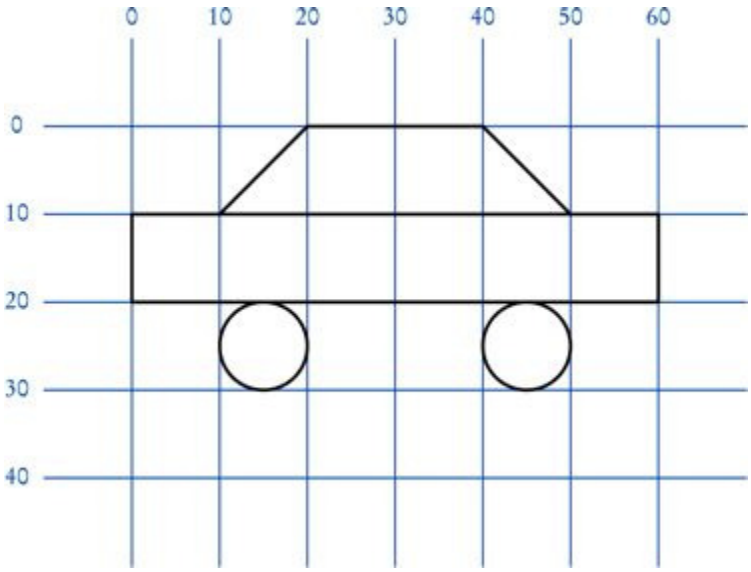
You will find the complete class definition at the end of this section. The `draw` method contains a rather long sequence of instructions for drawing the body, roof, and tires.

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 10).

The program that produces Figure 9 is composed of three classes.

Figure 10



Using Graph Paper to Find Shape Coordinates

- 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. 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:

```
Car car1 = new Car(0, 0);
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```

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.

ch03/car/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 }
```


ch03/car/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     /**
  13        Constructs a car with a given top-left
corner.
  14        @param x the x-coordinate of the
top-left corner
  15        @param y the y-coordinate of the
top-left corner
  16     */
  17     public Car(int x, int y)
  18     {
  19        xLeft = x;
  20        yTop = y;
  21     }
  22
  23     /**
  24        Draws the car.
  25        @param g2 the graphics context
  26     */
  27     public void draw(Graphics2D g2)
  28     {
  29        Rectangle body
  30              = new Rectangle(xLeft, yTop + 10,
60, 10);
  31        Ellipse2D.Double frontTire
  32              = new Ellipse2D.Double(xLeft +
10, yTop + 20, 10, 10);
  33        Ellipse2D.Double rearTire
  34              = new Ellipse2D.Double(xLeft +
40, yTop + 20, 10, 10);
  35
  36        // The bottom of the front windshield
  37        Point2D.Double r1
  38              = new Point2D.Double(xLeft + 10,
yTop + 10);
  39        // The front of the roof
  40        Point2D.Double r2
```

```
41                = new Point2D.Double(xLeft + 20,
   yTop);
42          // The rear of the roof
43          Point2D.Double r3
44                = new Point2D.Double(xLeft + 40,
   yTop);
45          // The bottom of the rear windshield
46          Point2D.Double r4
47                = new Point2D.Double(xLeft + 50,
   yTop + 10);
48
49          Line2D.Double frontWindshield
50                = new Line2D.Double(r1, r2);
51          Line2D.Double roofTop
52                = new Line2D.Double(r2, r3);
53          Line2D.Double rearWindshield
54                = new Line2D.Double(r3, r4);
55
56          g2.draw(body);
57          g2.draw(frontTire);
58          g2.draw(rearTire);
59          g2.draw(frontWindshield);
60          g2.draw(roofTop);
61          g2.draw(rearWindshield);
62       }
63
64       private int xLeft;
65       private int yTop;
66    }
```

ch03/car/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_(
12
```

```
 13          CarComponent component = new
CarComponent();
 14          frame.add(component);
 15
 16          frame.setVisible(true);
 17       }
 18  }
```

SELF CHECK

18. Which class needs to be modified to have the two cars positioned next to each other?

19. Which class needs to be modified to have the car tires painted in black, and what modification do you need to make?

20. How do you make the cars twice as big?

116

117

HOW TO 3.2: Drawing Graphical Shapes

You can write programs that display a wide variety of graphical shapes. These instructions give you a step-by-step procedure for decomposing a drawing into parts and implementing a program that produces the drawing.

Step 1 Determine the shapes that you need for the drawing.

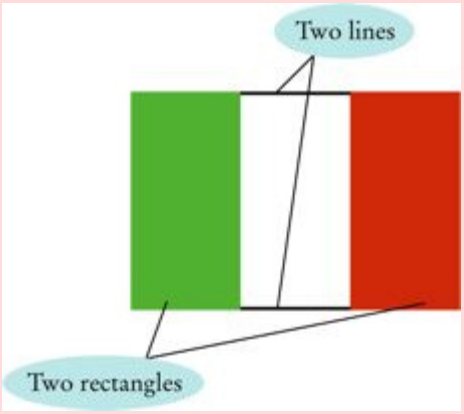
You can use the following shapes:

- 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 flag designs 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. 117

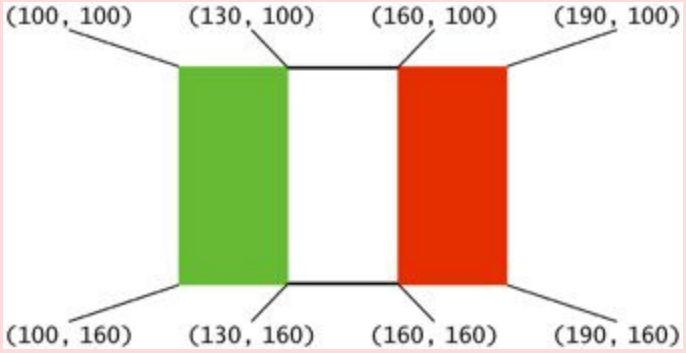
118

- 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 starting point and the end point.
- For text, you need the *x*- and *y*-positions 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);                    118
    Line2D.Double bottomLine = new Line2D.Double(           119
           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;
         // Your drawing code goes here
         . . .
    }
}
```

In our example, you can simply 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 and sizes. Specify the sizes in a constructor and supply a `draw` method:

```
    public class ItalianFlag
    {
         public ItalianFlag(double x, double y, double
    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);
             . . .
         }

         private int xLeft;
         private int yTop;
         private double width;
    }
```

119

120

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.

```
import javax.swing.*;
public class ItalianFlagViewer
{
    public static void main(String[] args)
    {
       JFrame frame = new JFrame();
       frame.setSize(300, 400);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOS
       ItalianFlagComponent component = new
ItalianFlagComponent();
       frame.add(component);
           frame.setVisible(true);
    }
}
```

120

121

RANDOM FACT 3.2: Computer Graphics

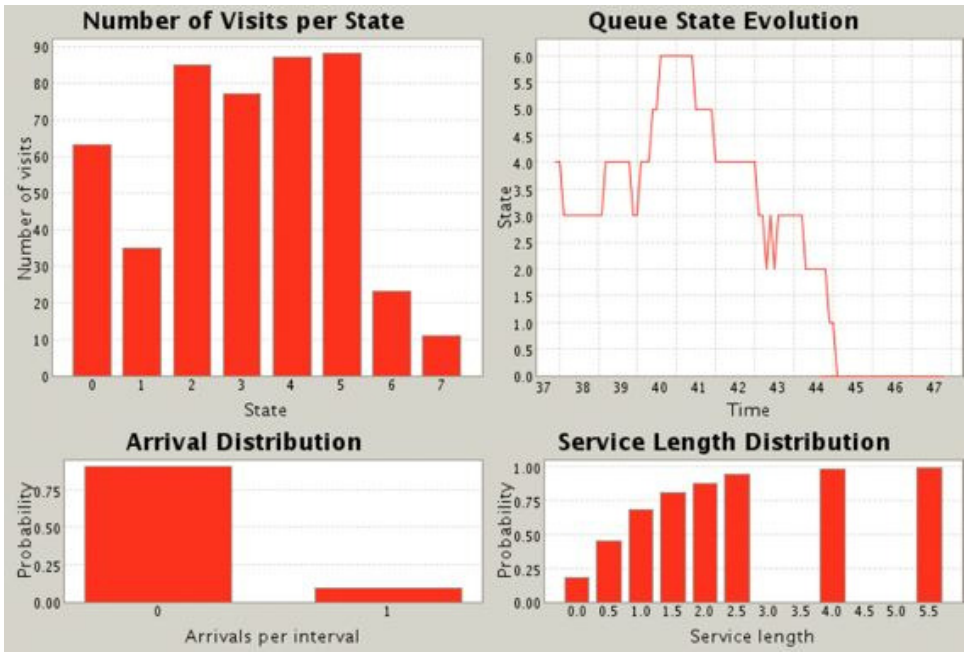
Generating and manipulating visual images is one of the most exciting applications of the computer. We distinguish different kinds of graphics.

Diagrams, such as numeric charts or maps, are artifacts that convey information to the viewer (see Diagrams figure). They do not directly depict anything that occurs in the natural world, but are a tool for visualizing information.

Scenes are computer-generated images that attempt to depict images of the real or an imagined world (see Scene figure). It turns out to be quite challenging to render light and shadows accurately. Special effort must be taken so that the images do not look too neat and simple; clouds, rocks, leaves, and dust in the real world have a complex and somewhat random appearance. The degree of realism in these images is constantly improving.

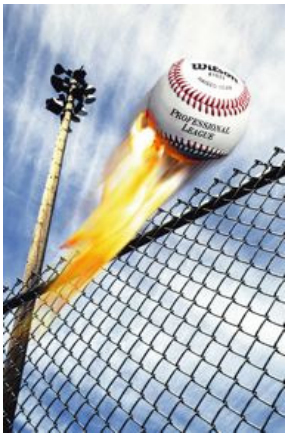
Manipulated images are photographs or film footage of actual events that have been converted to digital form and edited by the computer (see Manipulated Image figure). For example, film sequences in the movie *Apollo 13* were produced by starting from actual images and changing the perspective, showing the launch of the rocket from a more dramatic viewpoint.

Computer graphics is one of the most challenging fields in computer science. It requires processing of massive amounts of information at very high speed. New algorithms are constantly invented for this purpose. Displaying an overlapping set of three-dimensional objects



Diagrams

Scene



Manipulated Image

with curved boundaries requires advanced mathematical tools. Realistic modeling of textures and biological entities requires extensive knowledge of mathematics, physics, and biology.

122

123

CHAPTER SUMMARY

1. In order to implement a class, you first need to know which methods are required.
2. A method definition contains an access specifier (usually `public`), a return type, a method name, parameters, and the method body.
3. Constructors contain instructions to initialize objects. The constructor name is always the same as the class name.
4. Use documentation comments to describe the classes and public methods of your programs.
5. Provide documentation comments for every class, every method, every parameter, and every return value.
6. An object uses instance fields to store its state—the data that it needs to execute its methods.
7. Each object of a class has its own set of instance fields.
8. You should declare all instance fields as private.
9. Encapsulation is the process of hiding object data and providing methods for data access.
10. Constructors contain instructions to initialize the instance fields of an object.
11. Use the `return` statement to specify the value that a method returns to its caller.
12. A unit test verifies that a class works correctly in isolation, outside a complete program.

13. To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.
14. Instance fields belong to an object. Parameter variables and local variables belong to a method—they die when the method exits.
15. Instance fields are initialized to a default value, but you must initialize local variables.
16. The implicit parameter of a method is the object on which the method is invoked. The `this` reference denotes the implicit parameter.
17. Use of an instance field name in a method denotes the instance field of the implicit parameter.
18. It is a good idea to make a class for any part of a drawing that that can occur more than once.
19. To figure out how to draw a complex shape, make a sketch on graph paper.

123

124

FURTHER READING

1. http://verifiedvoting.org A site with information on voter-verifiable voting machines, founded by Stanford computer science professor David Dill.

REVIEW EXERCISES

★ **Exercise R3.1** Why is the `BankAccount(double initialBalance)` constructor not strictly necessary?

★ **Exercise R3.2** Explain the difference between

```
BankAccount b;
```

and

```
BankAccount b = new BankAccount(5000);
```

★ **Exercise R3.3** Explain the difference between

```
new BankAccount(5000);
```

and

```
BankAccount b = new BankAccount(5000);
```

★ **Exercise R3.4** What happens in our implementation of the `BankAccount` class when more money is withdrawn from the account than the current balance?

★ **Exercise R3.5** What is the value of `b.getBalance()` after the following operations?

```
BankAccount b = new BankAccount(10);
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
```

★★ **Exercise R3.6** If `b1` and `b2` refer to objects of class `BankAccount`, consider the following instructions.

```
b1.deposit(b2.getBalance());
b2.deposit(b1.getBalance());
```

Are the balances of `b1` and `b2` now identical? Explain.

★★ **Exercise R3.7** What is the `this` reference? Why would you use it?

★★ **Exercise R3.8** 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
}
```

124

125

★★ **Exercise R3.9** Suppose you want to implement a class `SavingsAccount`. A savings account has `deposit`, `withdraw`, and `getBalance` methods like a bank account, but it has a fixed interest rate that should be set in the constructor, together with the initial balance. An

`addInterest` method should be provided to add the earned interest to the account. This method should have no parameters since the interest rate is already known. It should have no return value since the new balance can be obtained by calling `getBalance`. Give the public interface for this class.

★★ **Exercise R3.10** What are the accessors and mutators of the `CashRegister` class?

★ **Exercise R3.11** Explain the difference between a local variable and a parameter variable.

★ **Exercise R3.12** Explain the difference between an instance field and a local variable.

★★G **Exercise R3.13** Suppose you want to write a program to show a suburban scene, with several cars and houses. Which classes do you need?

★★★G **Exercise R3.14** Explain why the calls to the `getWidth` and `getHeight` methods in the `CarComponent` class have no explicit parameter.

★★G **Exercise R3.15** How would you modify the `Car` class in order to show cars of varying sizes?

- Additional review exercises are available in Wiley PLUS.

PROGRAMMING EXERCISES

★ **Exercise P3.1.** 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.

★ **Exercise P3.2.** 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% interest
```

the balance in `momsSavings` is $1,100. Also supply a `BankAccountTester` class that prints the actual and expected balance.

125

126

★★ **Exercise P3.3.** Write a class `SavingsAccount` that is similar to the `BankAccount` class, except that it has an added instance field `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%. Then apply the `addInterest` method and print the resulting balance. Also compute the expected result by hand and print it.

★★ **Exercise P3.4.** Implement a class `Employee`. An employee has a name (a string) and a salary (a `double`). Provide a constructor with two parameters

```
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% raise
```

Supply an `EmployeeTester` class that tests all methods.

★★ **Exercise P3.5.** 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.

★★ **Exercise P3.6.** 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 latter, you also need to store the *number of quizzes* that the student took.

Supply a `StudentTester` class that tests all methods.

★ **Exercise P3.7.** 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.

126

127

★★ **Exercise P3.8.** 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 program `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.

★★ **Exercise P3.9.** 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()
```

127

128

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.

★★ **Exercise P3.10.** Implement a class Moth that models a moth flying across a straight line. The moth has a position, 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 void 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.

★ **Exercise P3.11.** 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` method simulates spraying with insecticide, which reduces the population by 10%. The `getRoaches` method returns the current number of roaches. A program called `RoachSimulation` simulates a population that starts out

with 10 roaches. Breed, spray, and print the roach count. Repeat three more times.

★★ **Exercise P3.12.** 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. Extra credit if your program gives the nod to your favored party if the votes are tallied after 8 p.m. on the first Tuesday in November, but acts normally on all other dates. (*Hint:* Use the `GregorianCalendar` class—see Programming Project 2.1.)

★★G **Exercise P3.13.** Draw a "bull's eye"—a set of concentric rings in alternating black and white colors.

Your program should be composed of classes `BullsEye`, `BullsEyeComponent`, and `BullsEyeViewer`.

128

129

★★G **Exercise P3.14.** 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.

★★G **Exercise P3.15.** Extend Exercise p3.14 by supplying a `House` constructor for specifying the position and size. Then populate your screen with a few houses of different sizes.

★★G **Exercise P3.16.** Change the car drawing program to make the cars appear in different colors. Each `Car` object should store its own color. Supply modified `Car` and `CarComponent` classes.

★★G **Exercise P3.17.** 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.

★★G **Exercise P3.18.** 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`.

★★G **Exercise P3.19.** Write a program that displays the Olympic rings. Color the rings in the Olympic colors.

Provide a class `OlympicRingViewer` and a class `OlympicRingComponent`.

★★G **Exercise P3.20.** Make a bar chart to plot the following data set. Label each bar. Make the bars horizontal for easier labeling.

Bridge Name	Longest Span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinac	3,800

129

130

- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 3.1.** 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. Produce a test program that verifies that the fees are calculated correctly over several months.

★★★ **Project 3.2.** 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. Use BlueJ to create two objects of this class and invoke their `sayHello` methods.

That is boring—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 4.6 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."`

Test your class in BlueJ. Make objects that greet the world and Dave, and invoke methods on them.

130

131

ANSWERS TO SELF-CHECK QUESTIONS

1. The programmers who designed and implemented the Java library.
2. Other programmers who work on the personal finance application.
3. `harrysChecking.withdraw(harrysChecking.getBalance())`
4. Add an `accountNumber` parameter to the constructors, and add a `getAccount-Number` method. There is no need for a `setAccountNumber` method—the account number never changes after construction.
5.

```
/**
    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
*/
```

6. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table.
7. An instance field

   ```
   private int accountNumber;
   ```

   needs to be added to the class.
8. You can't tell from the public interface, but the source file (which is a part of the JDK) contains these definitions:

```
private int x;
private int y;
private int width;
private int height;
```

9.

```
public int getWidth()
{
      return width;
}
```

10. 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;
}
```

11. One `BankAccount` object, no `BankAccountTester` object. The purpose of the `BankAccountTester` class is merely to hold the `main` method.

12. In those environments, you can issue interactive commands to construct `BankAccount` objects, invoke methods, and display their return values.

13. 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 call values; local variables must be explicitly initialized.

14. One instance field, named `balance`. Three local variables, one named `harrysChecking` and two named `newBalance` (in the `deposit` and `withdraw` methods); two parameter variables, both named `amount` (in the `deposit` and `withdraw` methods).

15. One implicit parameter, called `this`, of type `BankAccount`, and one explicit parameter, called `amount`, of type `double`.

16. It is not a legal expression. `this` is of type `BankAccount` and the `BankAccount` class has no field named `amount`.

17. No implicit parameter—the method is static—and one explicit parameter, called `args`.

18. `CarComponent`

19. In the `draw` method of the `Car` class, call

```
g2.fill(frontTire);
g2.fill(rearTire);
```

20. Double all measurements in the `draw` method of the `Car` class.