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.

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;
```

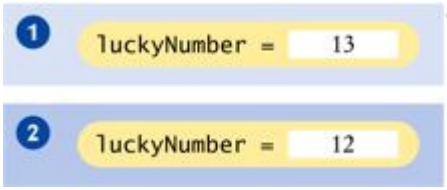
If you want to change the value of the variable, simply assign the new value:

```
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

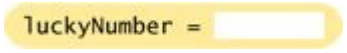


Assigning a New Value to a Variable

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!";?`

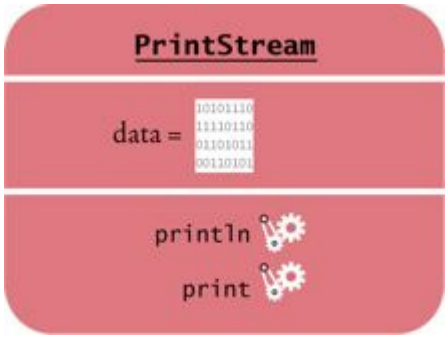
2.3 Objects, Classes, and Methods

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();
```

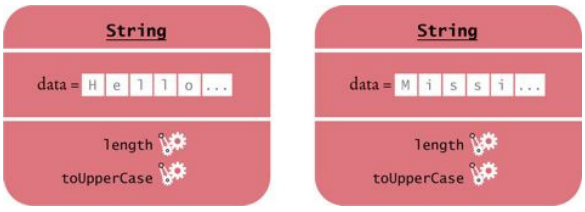
sets `n` to the number of characters in the `String` object `"Hello, World!"`. After the instructions in the length method are executed, `n` is set to 13. (The quotation marks are not part of the string, and the `length` method does not count them.)

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:

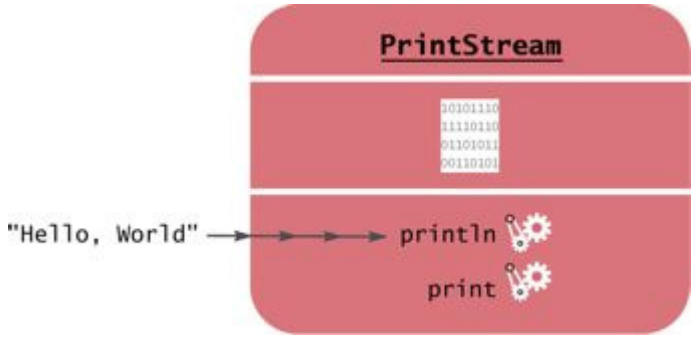
> The return value of a method is a result that the method has computed for use by the code that called it.

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

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

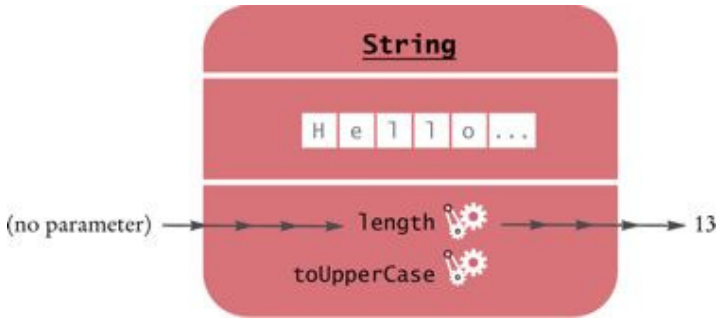
```
System.out.println(greeting.length());
```

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

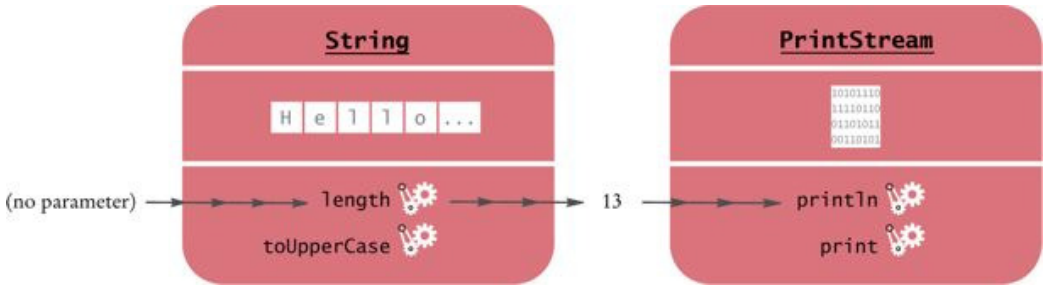
```
river.replace("issipp", "our")
```

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

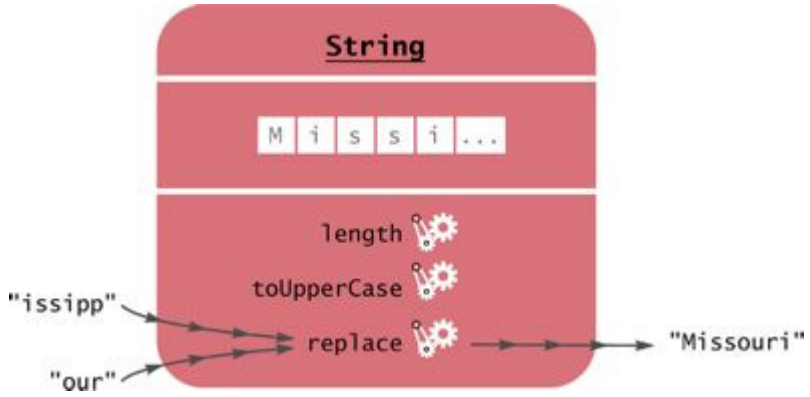
- one implicit parameter: the string `"Mississippi"`
- two explicit parameters: the strings `"issipp"` and `"our"`
- a return value: the string `"Missouri"`

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
```

43

44

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`.

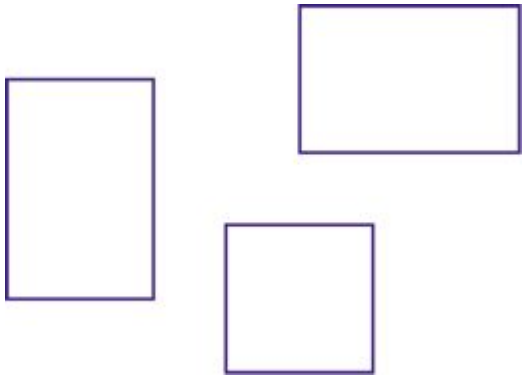
2.6 Constructing Objects

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.

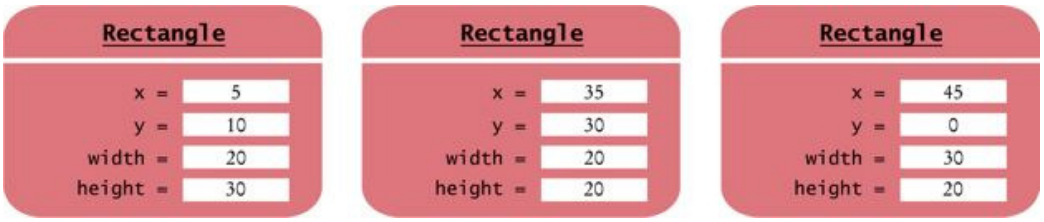
Note that a `Rectangle` object isn't a rectangular shape—it is an object that contains a set of numbers. The numbers *describe* the rectangle (see Figure 10). Each rectangle is described by the *x*- and *y*-coordinates of its top-left corner, its width, and its height.

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);
```

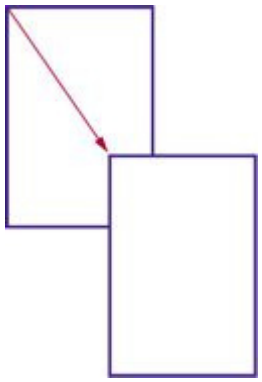
moves the rectangle by 15 units in the *x*-direction and 25 units in the *y*-direction (see Figure 11). Moving a rectangle doesn't change its width or height, but it changes the top-left corner. Afterwards, the top-left corner is at (20, 35).

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
```

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?

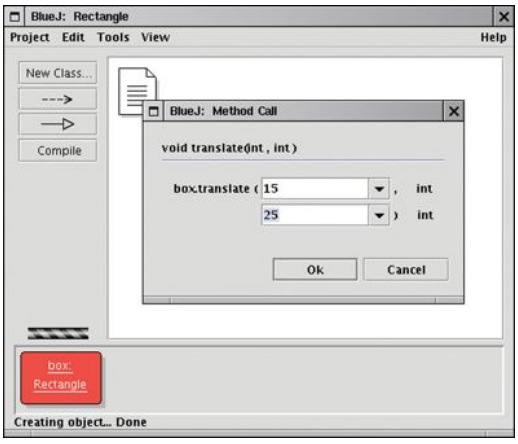
21. Why doesn't the `MoveTester` program need to print the width and height of the rectangle?

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.

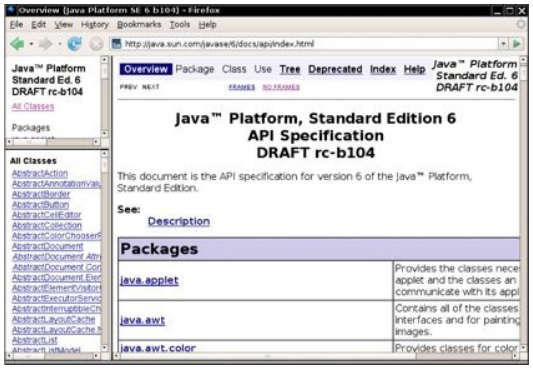
The API documentation documents all classes in the Java library—there are thousands of them (see Figure 12). Most of the classes are rather specialized, and only a few are of interest to the beginning programmer.

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).

50

51

Figure 12



The API Documentation of the Standard Java Library

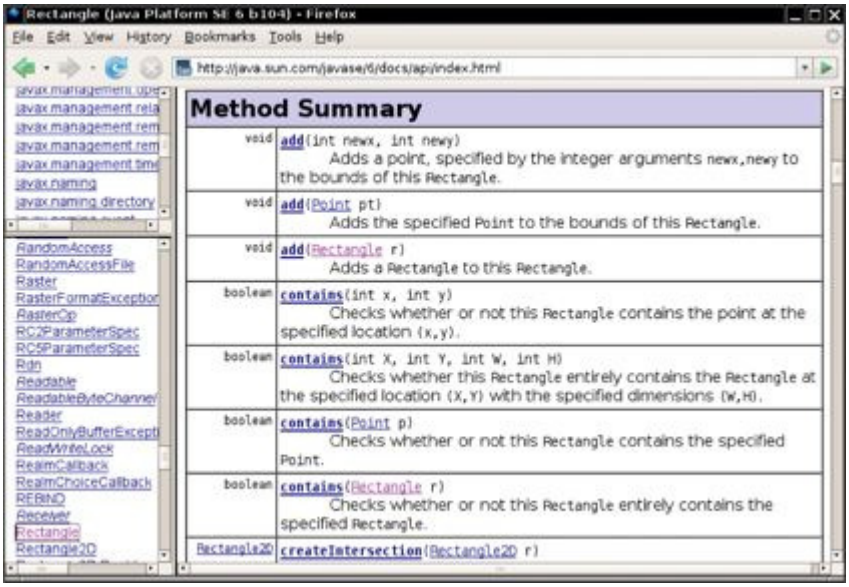
Figure 13

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.

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

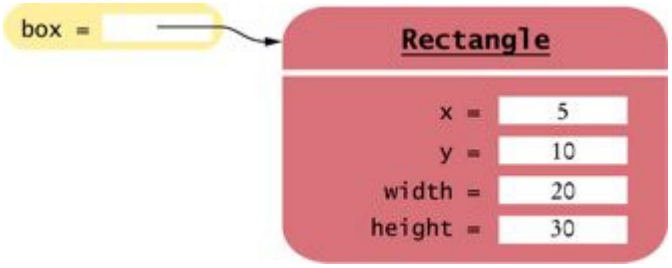
In Java, a variable whose type is a class does not actually hold an object. It merely holds the memory *location* of an object. The object itself is stored elsewhere—see Figure 16.

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.

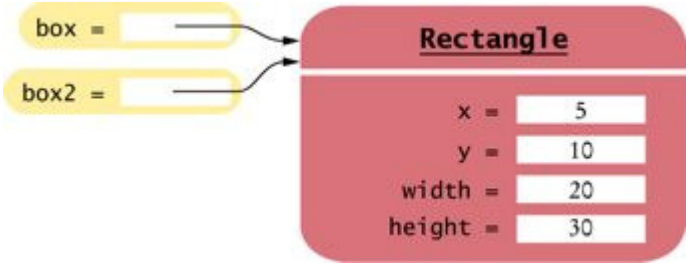
```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Figure 16



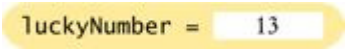
An Object Variable Containing an Object Reference

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.

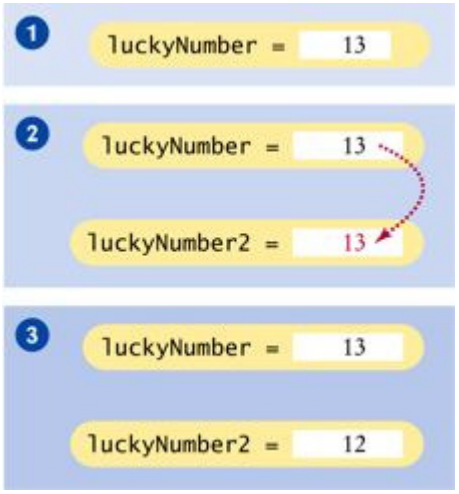
```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

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

54

55

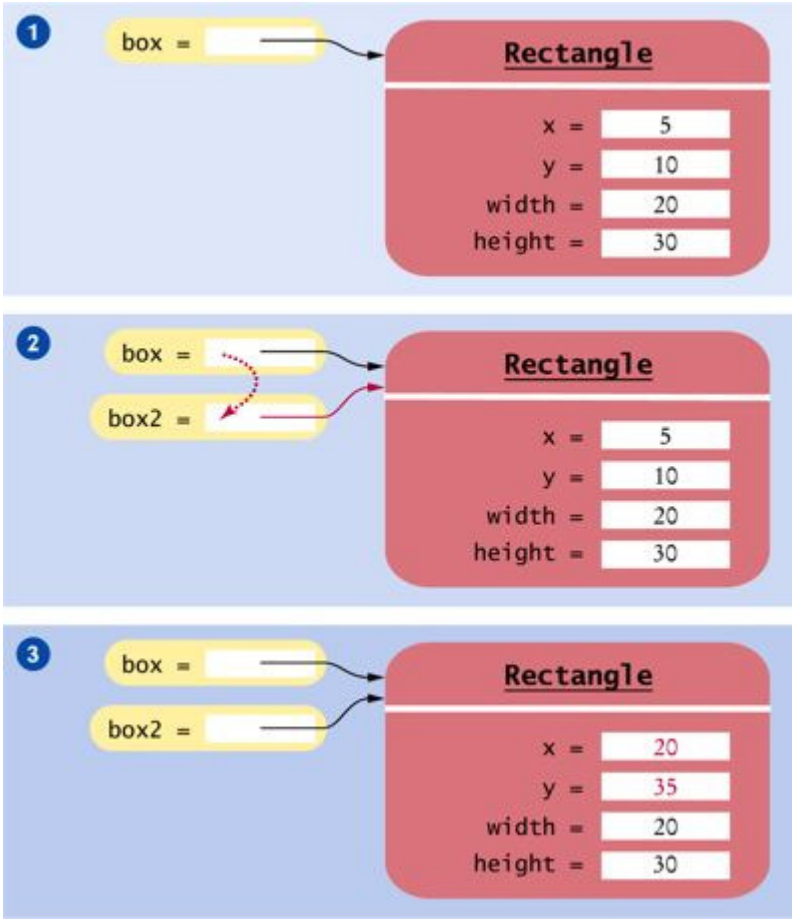
Figure 19



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.

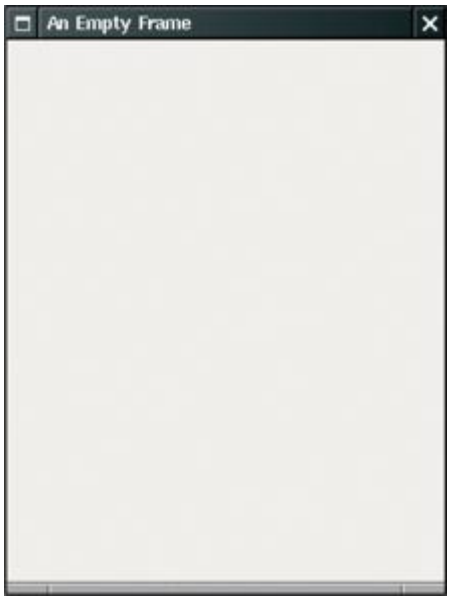
```
frame.setTitle("An Empty Frame");
```

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?

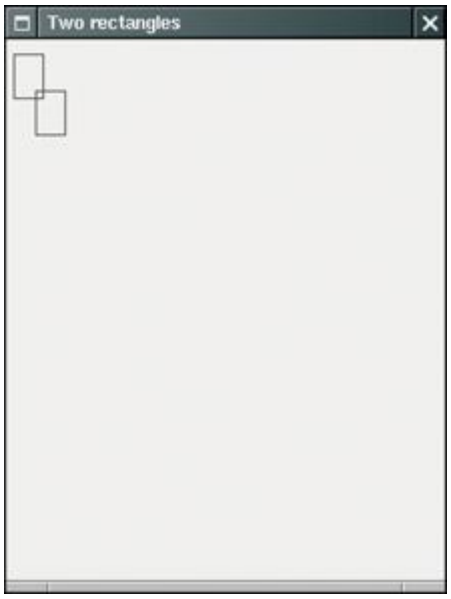
2.12 Drawing on a Component

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          {
13                // Prepare for extended graphics
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

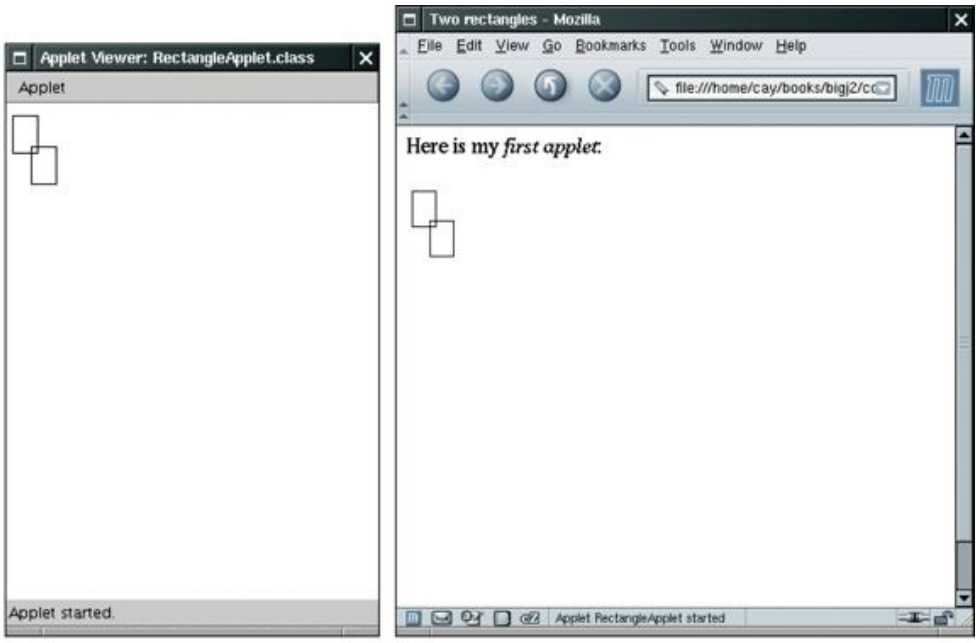
```
 1  <html>
 2        <head>
 3           <title>Two rectangles</title>
 4        </head>
 5        <body>
 6           <p>Here is my <i>first
applet</i>:</p>
 7           <applet code="RectangleApplet.class"
width="300" height="400">
 8           </applet>
 9        </body>
10  </html>
```

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.

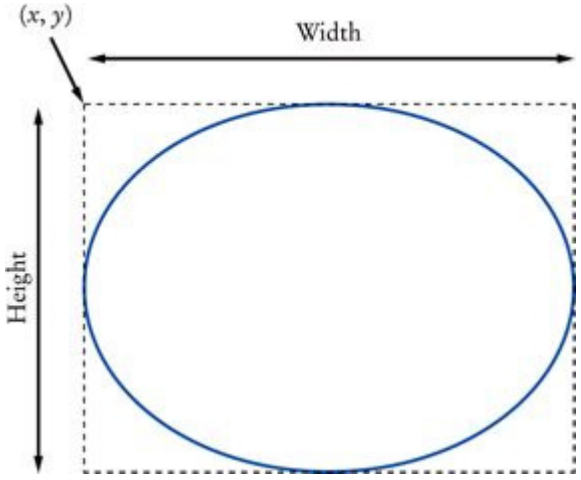
To draw an ellipse, you specify its bounding box (see Figure 23) in the same way that you would specify a rectangle, namely by the *x*- and *y*-coordinates of the top-left corner and the width and height of the box.

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

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 {
```

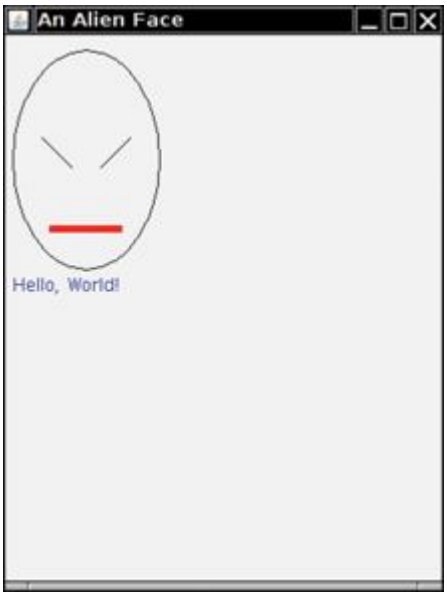
```
15          public void paintComponent(Graphics g)
16          {
17             // Recover Graphics2D
18             Graphics2D g2 = (Graphics2D) g;
19
20             // Draw the head
21             Ellipse2D.Double head = new
Ellipse2D.Double(5, 10, 100, 150);
22             g2.draw(head);
23
24             // 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

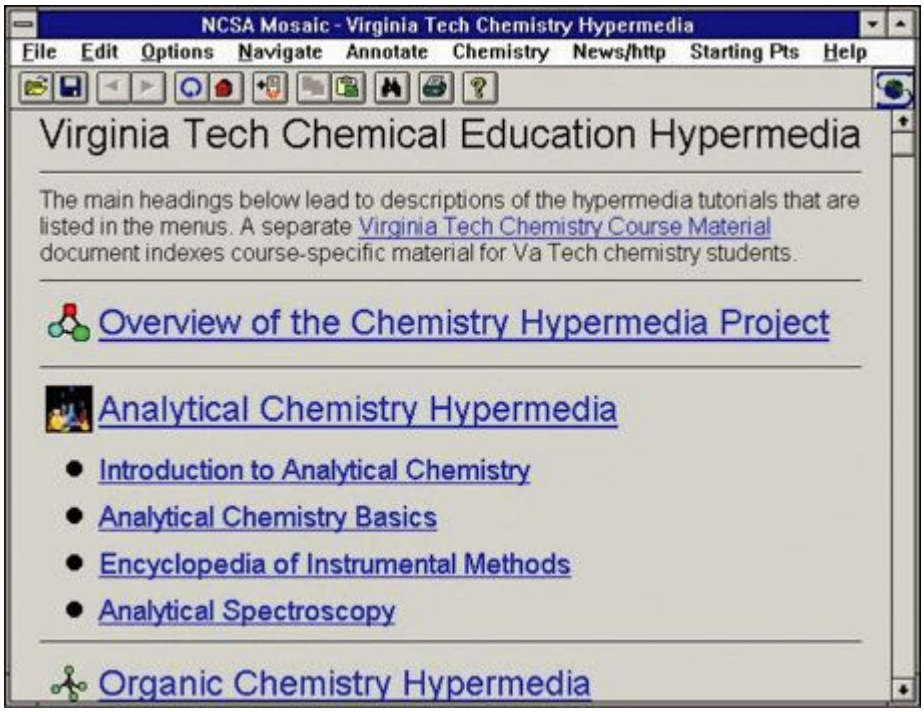
In 1962, J.C.R. Licklider was head of the first computer research program at DARPA, the Defense Advanced Research Projects Agency. He wrote a series of papers describing a "galactic network" through which computer users could access data and programs from other sites. This was well before computer networks 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

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?

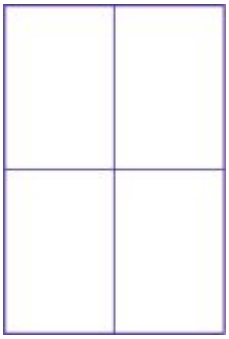
Additional review exercises are available in WileyPLUS.

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.

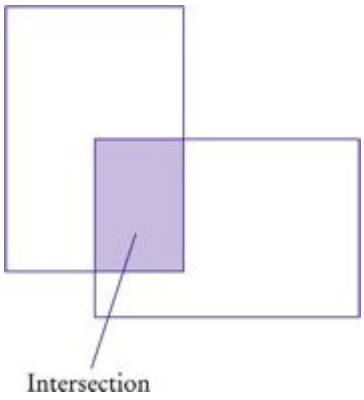
★★ **Exercise P2.3.** Write a program called `FourRectanglePrinter` that constructs a `Rectangle` object, prints its location by calling `System.out.println(box)`, and then translates and prints it three more times, so that, if the rectangles were drawn, they would form one large rectangle:



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.

79

80

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));
```