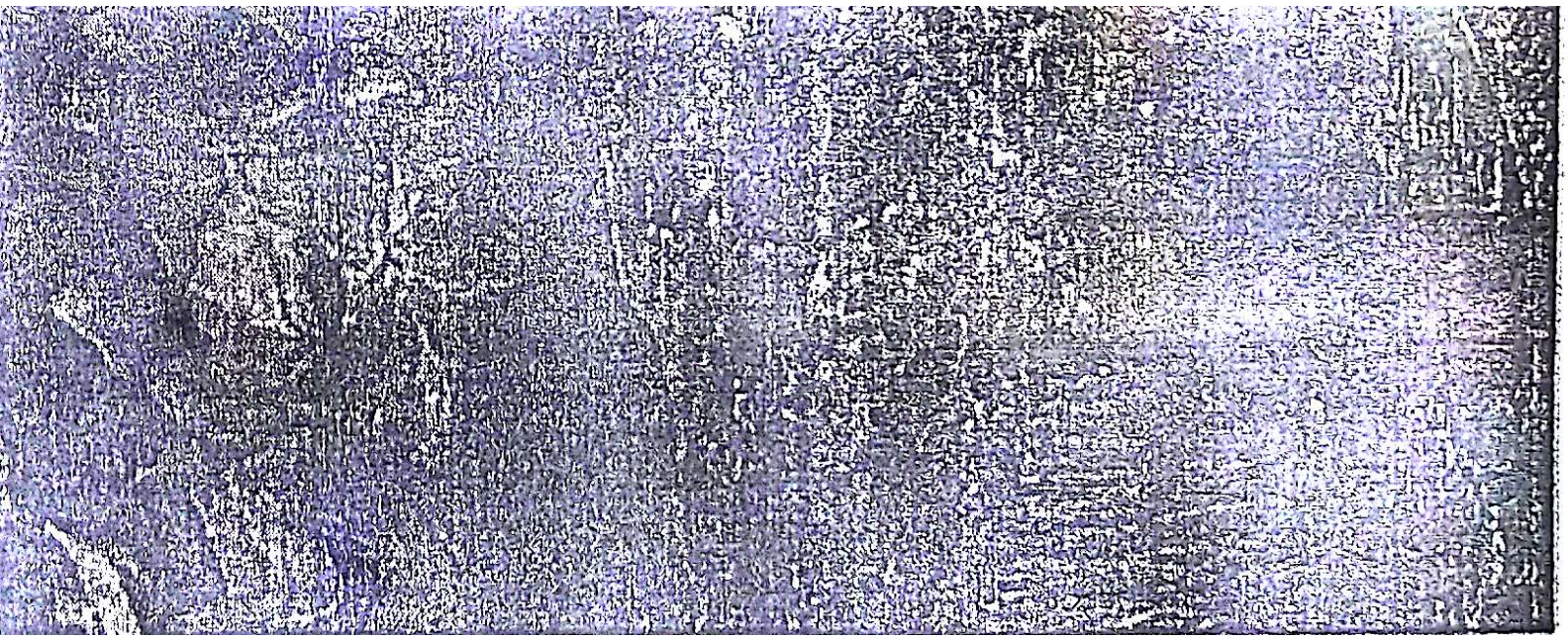

Scanned by CamScanner

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.

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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;
```

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

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.

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)
```

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.

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

Scanned by CamScanner

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`.

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:

- 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. ***Self Check 2.1 What is the Type of the Values 0 and "0"?***

   What is the type of the values 0 and "0"?

   Answer

2. ***Self Check 2.2 Which of the Following are Legal Identifiers?***

   Which of the following are legal identifiers?

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

3. ***Self Check 2.3 Define a Variable to Hold your Name***

   Answer

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

Scanned by CamScanner

Answer

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

Syntax 2.1: Variable Definition

syntax 2.1

typeName variableName = value;
or
typeName variableName;

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

Purpose

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

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

2.2 The Assignment Operator

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

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

```
int luckyNumber = 13;
```
(1)

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

```
luckyNumber = 12;
```
(2)

The assignment replaces the original value of the variable (see Figure 2-1).

Scanned by CamScanner

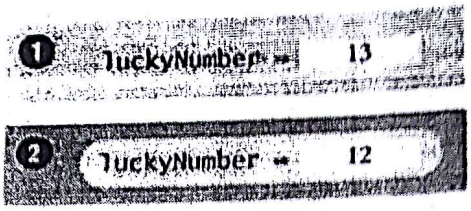

FIGURE 2-1 Assigning a New Value to a Variable

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.

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 2-2).

All variables must be initialized before you access them.

FIGURE 2-2 An Uninitialized Object Variable

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

```
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
```


Animation: Variable Initialization and Assignment

Self Check

4. *Self Check 2.4 Is 12 = 12 a Valid Expression?*

Is `12 = 12` a valid expression in the Java language?

5. *Self Check 2.5 Change Greeting to "Hello, Nina!"*

Answer

How do you change the value of the `greeting` variable to `"Hello, Nina!"`?

Answer

Scanned by CamScanner

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

Syntax 2.2: Assignment

syntax 2.2

```
variableName = value;
```

```
luckyNumber = 12;
```

Purpose

To assign a new value to a previously defined variable

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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.

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

Scanned by CamScanner

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 2-3 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.

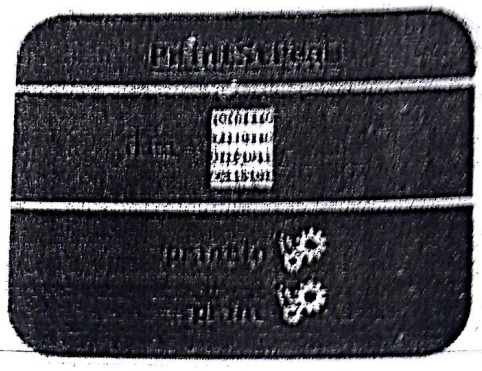

FIGURE 2-3 Representation of the `System.out` Object

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.

A class defines 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.

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.

In section 2.4, 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();
```

Scanned by CamScanner

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

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.

Figure 2-4 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.

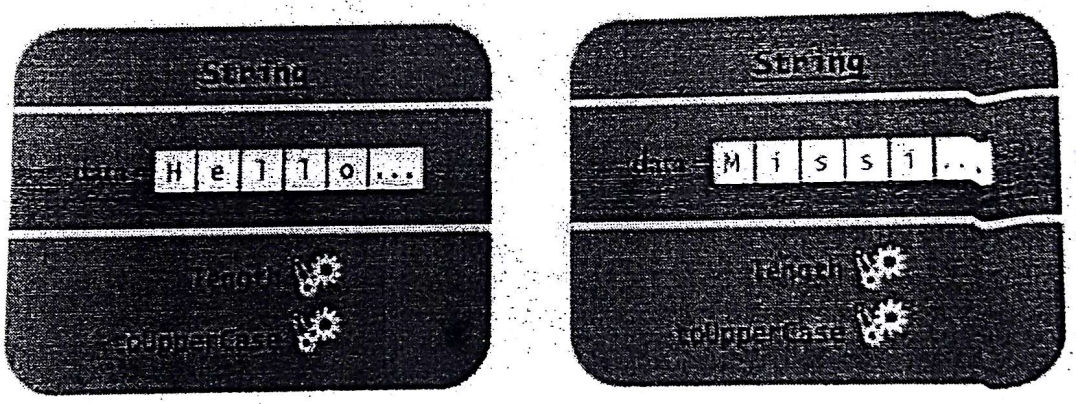

FIGURE 2-4 A Representation of Two `String` Objects

Self Check

6. ***Self Check 2.6 Compute the Length of "Mississippi"***

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

   Answer

7. ***Self Check 2.7 Print the Uppercase Version of "Hello, World!"***

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

   Answer

8. ***Self Check 2.8 Is it Legal to Call `river.println()`?***

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

Scanned by CamScanner

Answer

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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 `System.out.println(greeting)`

Figure 2-5 illustrates passing of the parameter to the method.

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

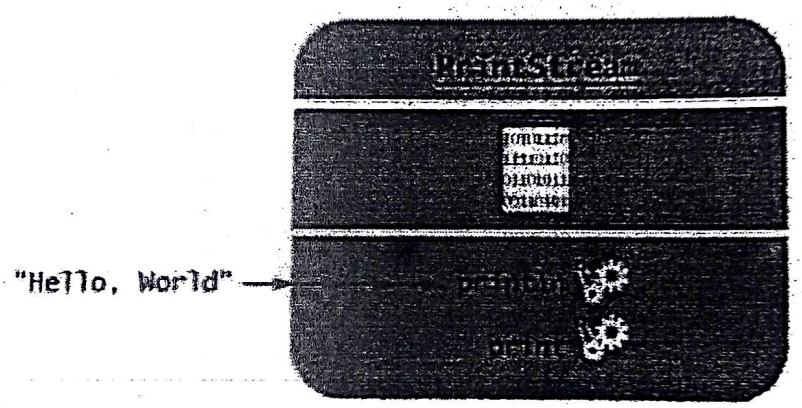

FIGURE 2-5 Passing a Parameter to the `println` Method

A parameter is an input to a 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 `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 2-6). 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 return value of a method is a result that the method has computed for use by the code that called it.

Scanned by CamScanner

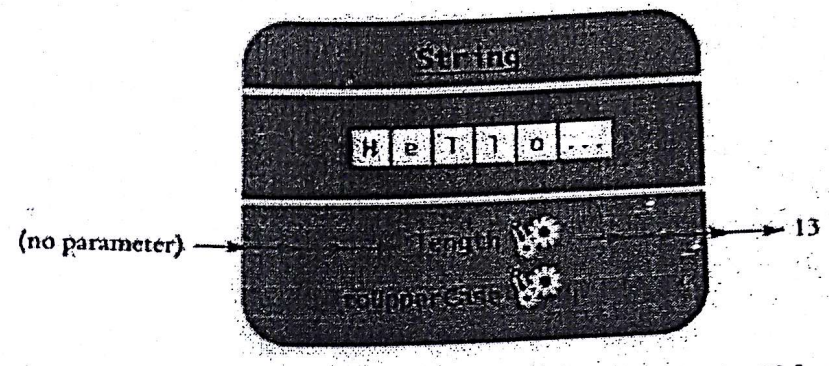

FIGURE 2-6 Invoking the `length` Method on a `String` Object

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:

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

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

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

Animation: Parameter Passing

The method call `greeting.length()` returns a value—the integer 13. The return value becomes a parameter of the `println` method. Figure 2-7 shows the process.

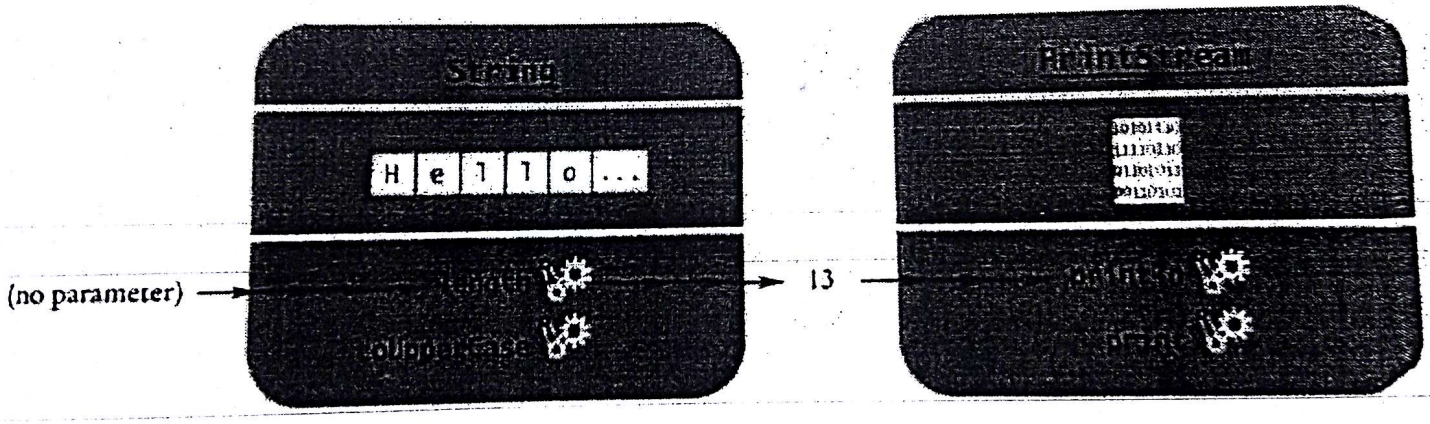

FIGURE 2-7 Passing the Result of a Method Call to Another Method

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

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

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

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

As Figure 2-8 shows, this method call has

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

Scanned by CamScanner

- a return value: the string "Missouri"

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.)

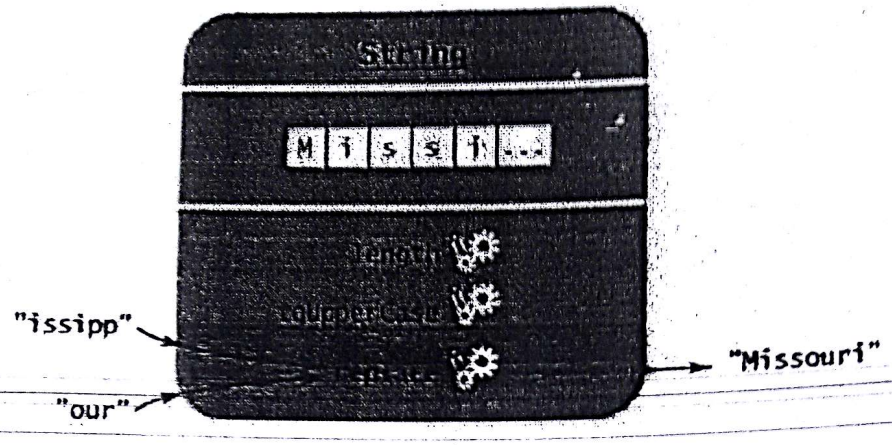

FIGURE 2-8 Calling the `replace` Method

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)
```

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

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

```
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. ***Self Check 2.9 Implicit/Explicit Parameters, Return Value in Call `river.length()`***

   What are the implicit parameters, explicit parameters, and return values in the method call `river.length()`?

Scanned by CamScanner

Answer

10. *Self Check 2.10 Result of Call* `river.replace("p", "s")`

What is the result of the call `river.replace("p", "s")`?

Answer

11. *Self Check 2.11 Result of Call* `greeting.replace("World", "Dave").length()`

What is the result of the call `greeting.replace("World", "Dave").length()`?

Answer

12. *Self Check 2.12 How is* `toUpperCase` *Defined in the* `String` *Class?*

How is the `toUpperCase` method defined in the `String` class?

Answer

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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 `En` 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 floatingpoint 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.

Scanned by CamScanner

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 × *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. ***Self Check 2.13 Number Type for Storing the Area of a Circle***

    Which number type would you use for storing the area of a circle?

    Answer

14. ***Self Check 2.14 Why is the Expression 13.println() an Error?***

    Why is the expression `13.println()` an error?

    Answer

15. ***Self Check 2.15 Expression for Average of the Values x and y***

    Write an expression to compute the average of the values `x` and `y`.

    Answer

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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 2-9. These objects are useful for a variety of purposes.

Scanned by CamScanner

You can assemble rectangles into bar charts, and you can program simple games by moving rectangles inside a window.

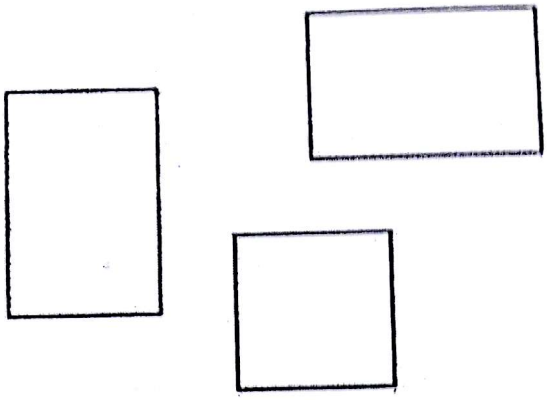

FIGURE 2-9 Rectangular Shapes

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 2-10). Each rectangle is described by the *x*- and *y*-coordinates of its top-left corner, its width, and its height.

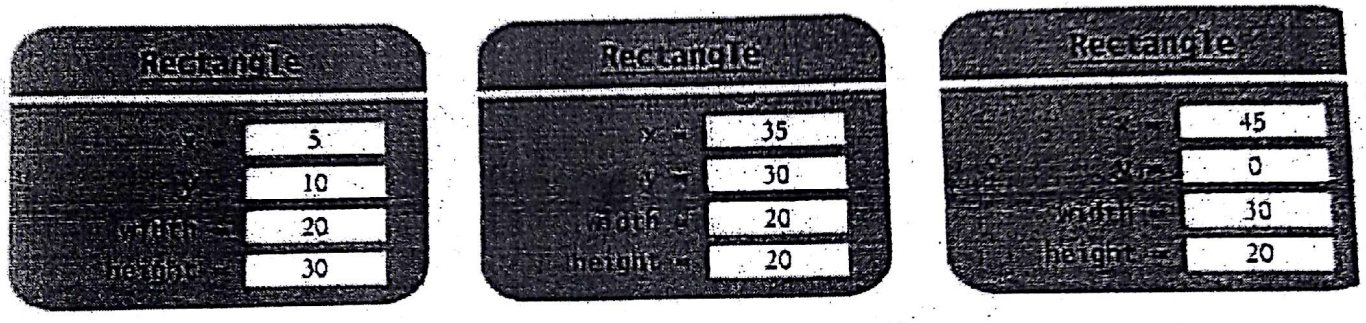

FIGURE 2-10 `Rectangle` Objects

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

Use the new operator, followed by a class name and 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);
```

Scanned by CamScanner

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.

Self Check

16. ***Self Check 2.16 Construct a Square with Center (100,100) and Side Length 20***

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

Answer

17. ***Self Check 2.17 Output of `System.out.println(new Rectangle().getWidth())`***

What does the following statement print?

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

Answer

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

Syntax 2.3: Object Construction

syntax 2.3

```
new ClassName(parameters)
```

```
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

Scanned by CamScanner

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

Common Error 2.1: Trying to Invoke a Constructor Like a Method

Common Error 2.1

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
```

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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 2-11). 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).

Scanned by CamScanner

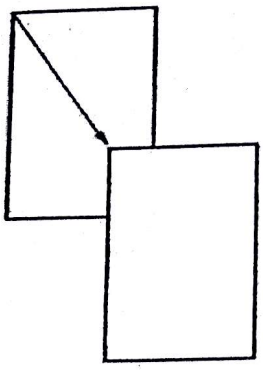

FIGURE 2-11 Using the `translate` Method to Move a Rectangle

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

Self Check

18. ***Self Check 2.18 Is `String.toUpperCase` an Accessor or a Mutator?***

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

Answer

19. ***Self Check 2.19 Move Box so that its Top-Left Corner is the Origin (0, 0)***

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

Answer

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.

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.

Scanned by CamScanner