33

Chapter 2 Using Objects

CHAPTER GOALS

- To learn about variables
- To understand the concepts of classes and objects
- To be able to call methods
- To learn about parameters and return values

T To implement test programs

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

G To write programs that display simple shapes

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

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

33

34

2.1 Types and Variables

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

In Java, every value has a type.

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

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

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

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

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

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

34

35

SYNTAX 2.1 Variable Definition

typeName variableName = value;

or

typeName variableName;

Example:

```
String greeting = “Hello, Dave!”;
```

Purpose:

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

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

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

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

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

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

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

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

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

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

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

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

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

These are firm rules of the Java language. If you violate one of them, the compiler will report an error. Moreover, there are a couple of *conventions* that you should follow so that other programmers will find your programs easy to read: 35

36

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

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

SELF CHECK

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

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

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

2.2 The Assignment Operator

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

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

```
int luckyNumber = 13;
```

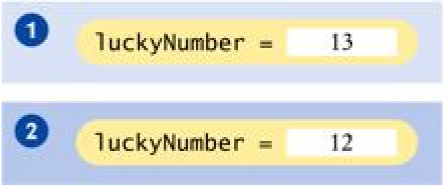
If you want to change the value of the variable, simply assign the new value:

```
luckyNumber = 12;
```

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

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

Figure 1



Assigning a New Value to a Variable

36

37

Figure 2

luckyNumber =

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!";`?

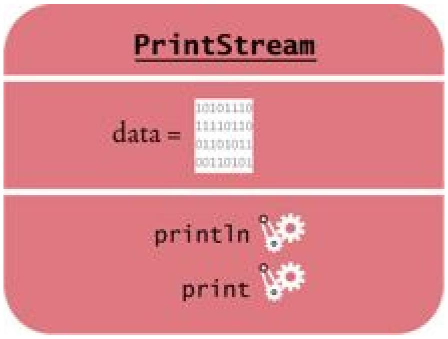
2.3 Objects, Classes, and Methods

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 37
38 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();
```

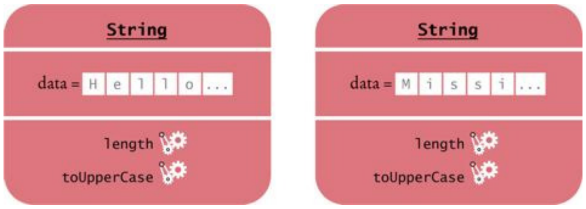
sets n to the number of characters in the String object “Hello, World!”. After the instructions in the length method are executed, n is set to 13. (The quotation marks are not part of the string, and the length method does not count them.)

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:

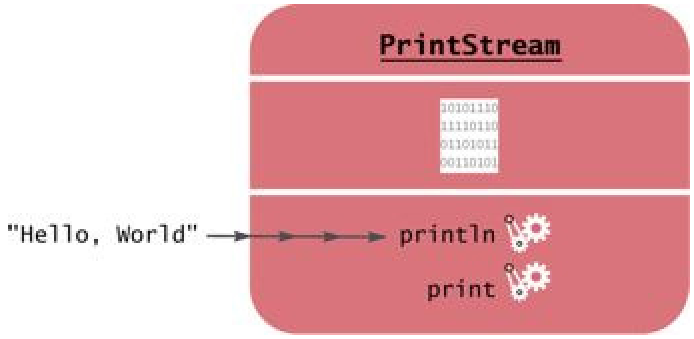
The return value of a method is a result that the method has computed for use by the code that called it.

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

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

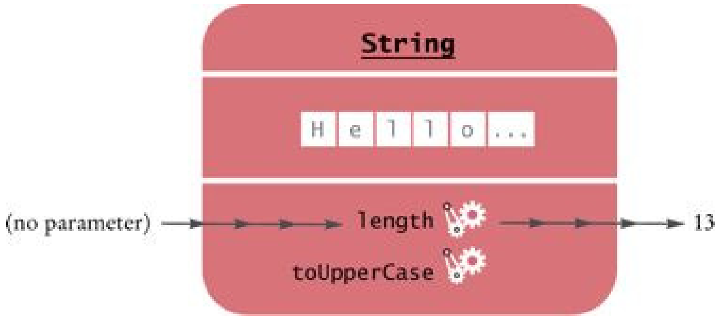
```
System.out.println(greeting.length());
```

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

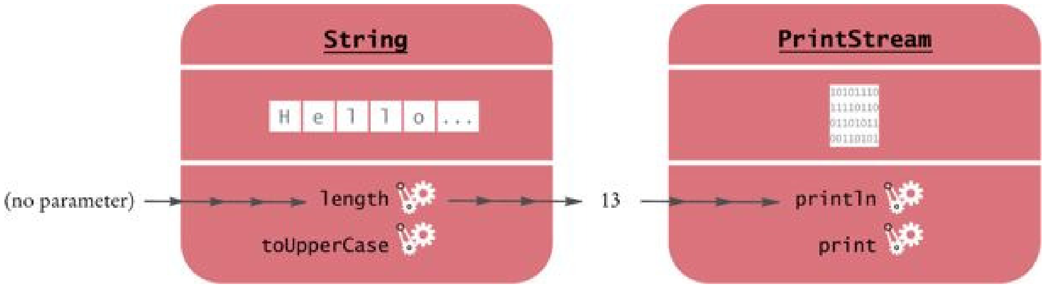
```
river.replace(“issipp”, “our”)
```

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”`

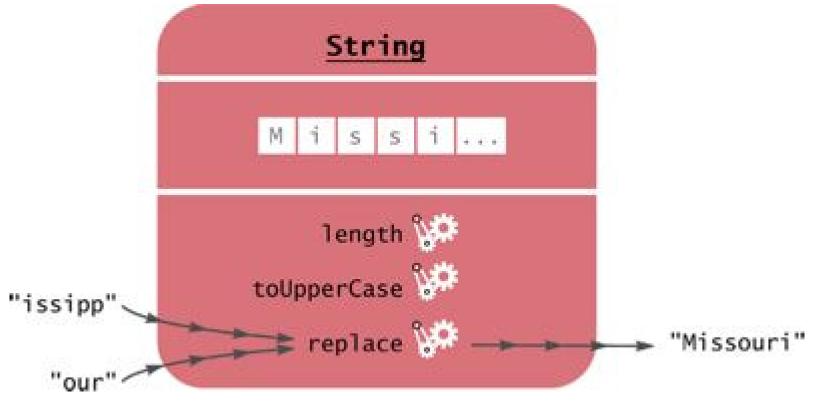
Figure 7



Passing the Result of a Method Call to Another Method

41

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")?

42

43

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

12. How is the toUpperCase method defined in the String class?

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 "×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 × *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.

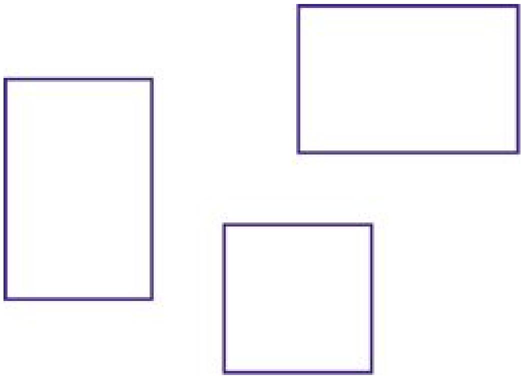
2.6 Constructing Objects

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.

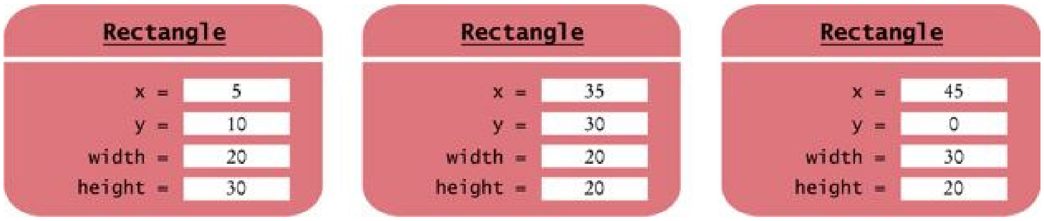
Note that a Rectangle object isn't a rectangular shape—it is an object that contains a set of numbers. The numbers *describe* the rectangle (see Figure 10). Each rectangle is described by the *x*- and *y*-coordinates of its top-left corner, its width, and its height.

Figure 9



Rectangular Shapes

Figure 10



Rectangle
x = 5
y = 10
width = 20
height = 30

Rectangle
x = 35
y = 30
width = 20
height = 20

Rectangle
x = 45
y = 0
width = 30
height = 20

`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

45

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,

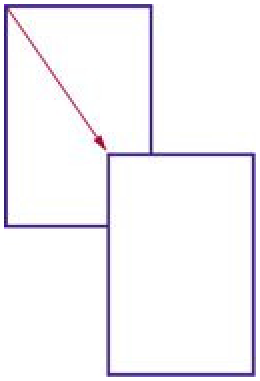
```
box.translate(15, 25);
```

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

```
import java.awt.Rectangle;

public class MoveTester
{
     public static void main(String[] args)
     {
          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”);

          System.out.print(“y: ”);
          System.out.println(box.getY());
          System.out.println(“Expected: 35”);
     }
}
```

Output

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

48

49

For this program, we needed to carry out another step: We needed to *import* the `Rectangle` class from a *package*. A package is a collection of classes with a related purpose. All classes in the standard library are contained in packages. The `Rectangle` class belongs to the package `java.awt` (where `awt` is an abbreviation for "Abstract Windowing Toolkit"), which contains many classes for drawing windows and graphical shapes.

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

To use the `Rectangle` class from the `java.awt` package, simply place the following line at the top of your program:

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

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

SYNTAX 2.4 Importing a Class from a Package

```
import packageName.ClassName;
```

Example:

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

Purpose

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

Self Check

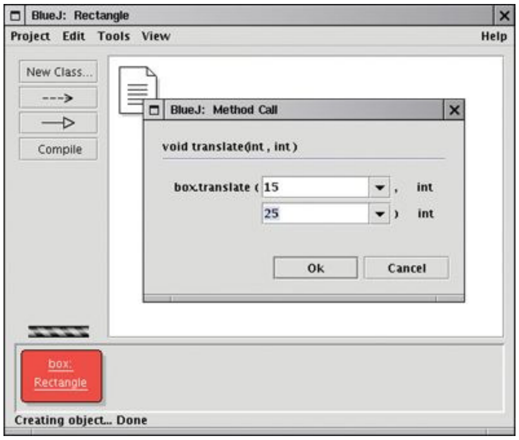
20. Suppose we had called `box.translate(25, 15)` instead of `box.translate(15, 25)`. What are the expected outputs?

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

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

Advanced Topic 2.1: Testing Classes in an Interactive Environment

Some development environments are specifically designed to help students explore objects without having to provide tester classes. These environments can be very helpful for gaining insight into the behavior of objects, and for promoting object-oriented thinking. The BlueJ environment (shown in Testing a Method Call in BlueJ) displays objects as blobs on a workbench. You can construct new objects, 49
50 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].



void translate(int , int)

box.translate (15 , int
25) int

Creating object... Done

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.

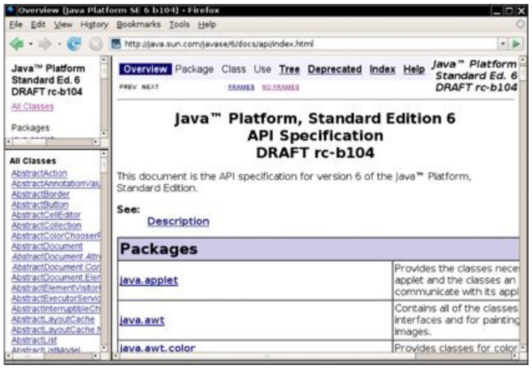
> The API (Application Programming Interface) documentation lists the classes and methods of the Java library.

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



Java™ Platform, Standard Edition 6
API Specification
DRAFT rc-b104

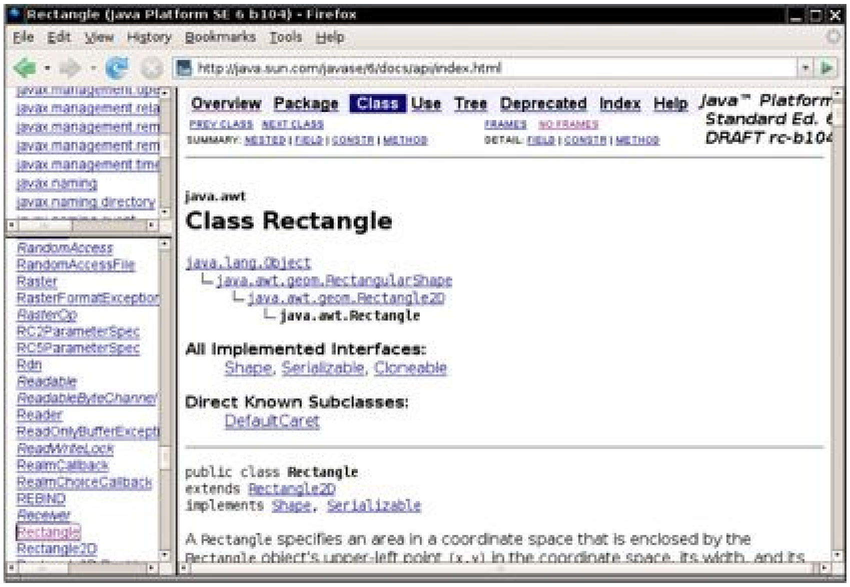
This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

See:
Description

Packages	
java.applet	Provides the classes nece applet and the classes an communicate with its appl
java.awt	Contains all of the classes interfaces and for painting images.
java.awt.color	Provides classes for color

The API Documentation of the Standard Java Library

Figure 13



java.awt

Class Rectangle

```
java.lang.Object
  └ java.awt.geom.RectangularShape
      └ java.awt.geom.Rectangle2D
          └ java.awt.Rectangle
```

All Implemented Interfaces:
Shape, Serializable, Cloneable

Direct Known Subclasses:
DefaultCaret

```
public class Rectangle
extends Rectangle2D
implements Shape, Serializable
```

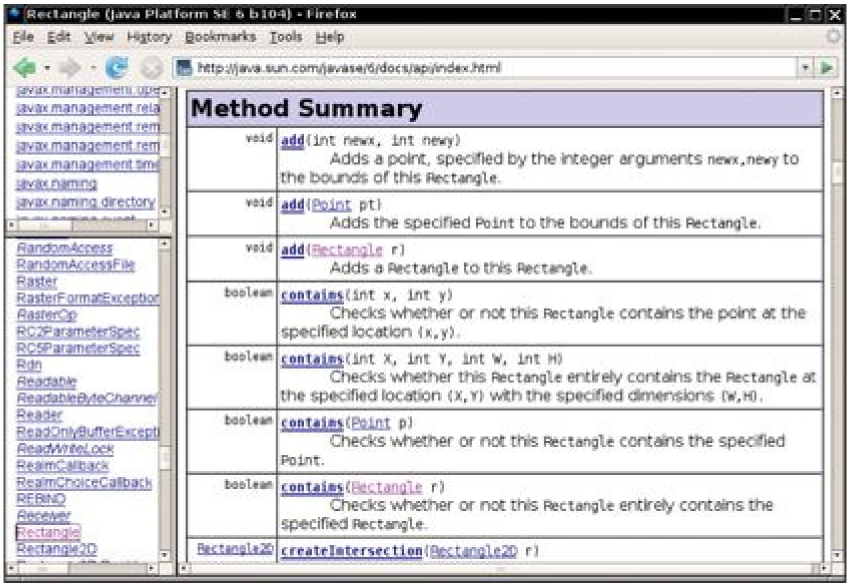
A Rectangle specifies an area in a coordinate space that is enclosed by the Rectangle object's upper-left point (x,y) in the coordinate space, its width, and its

The API Documentation for the `Rectangle` Class

The API documentation for each class starts out with a section that describes the purpose of the class. Then come summary tables for the constructors and methods (see Figure 14). Click on the link of a method to get a detailed description (see Figure 15).

As you can see, the `Rectangle` class has quite a few methods. While occasionally intimidating for the beginning programmer, this is a strength of the standard library. If you ever need to do a computation involving rectangles, chances are that there is a method that does all the work for you. 51

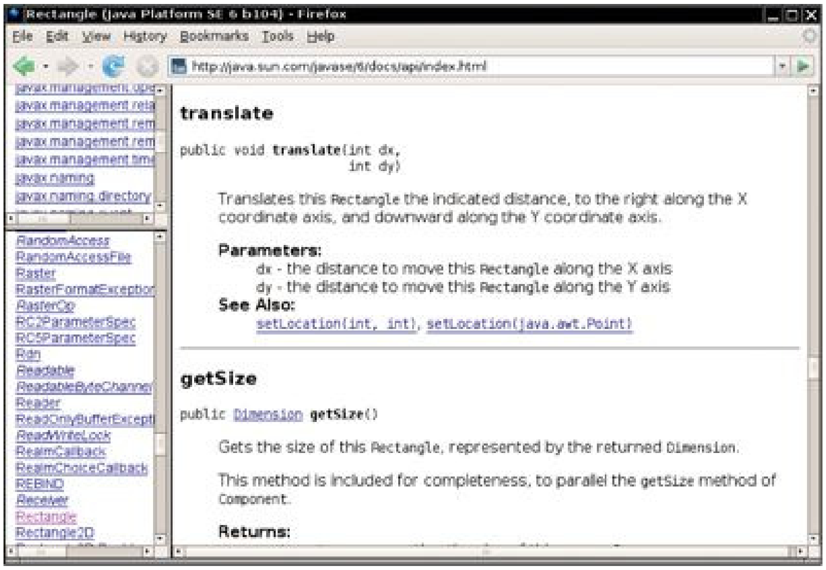
Figure 14



Method Summary	
void	add(int newx, int newy) Adds a point, specified by the integer arguments newx,newy to the bounds of this Rectangle.
void	add(Point pt) Adds the specified Point to the bounds of this Rectangle.
void	add(Rectangle r) Adds a Rectangle to this Rectangle.
boolean	contains(int x, int y) Checks whether or not this Rectangle contains the point at the specified location (x,y).
boolean	contains(int X, int Y, int W, int H) Checks whether this Rectangle entirely contains the Rectangle at the specified location (X,Y) with the specified dimensions (W,H).
boolean	contains(Point p) Checks whether or not this Rectangle contains the specified Point.
boolean	contains(Rectangle r) Checks whether or not this Rectangle entirely contains the specified Rectangle.
Rectangle2D	createIntersection(Rectangle2D r)

The Method Summary for the `Rectangle` Class

Figure 15



translate

```
public void translate(int dx,
                      int dy)
```

Translates this Rectangle the indicated distance, to the right along the X coordinate axis, and downward along the Y coordinate axis.

Parameters:
dx - the distance to move this Rectangle along the X axis
dy - the distance to move this Rectangle along the Y axis
See Also:
setLocation(int, int), setLocation(java.awt.Point)

getSize

```
public Dimension getSize()
```

Gets the size of this Rectangle, represented by the returned Dimension.

This method is included for completeness, to parallel the getSize method of Component.

Returns:

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

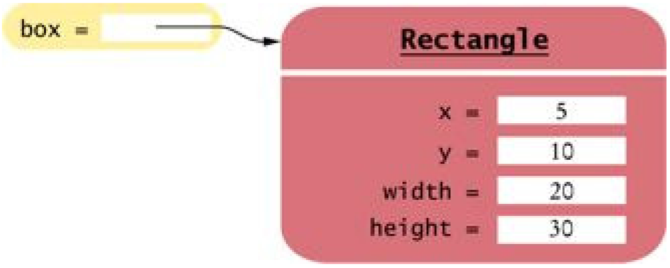
In Java, a variable whose type is a class does not actually hold an object. It merely holds the memory *location* of an object. The object itself is stored elsewhere—see Figure 16.

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

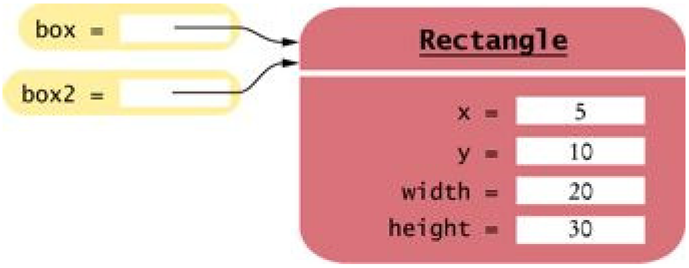


An Object Variable Containing an Object Reference

53

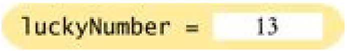
54

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;
```

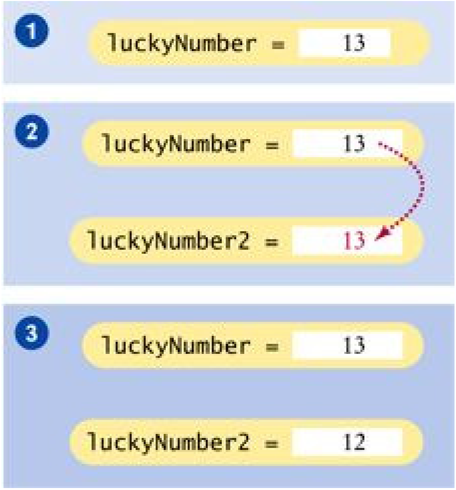
Now the variable `luckyNumber` contains the value 13, and `luckyNumber2` contains 12.

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

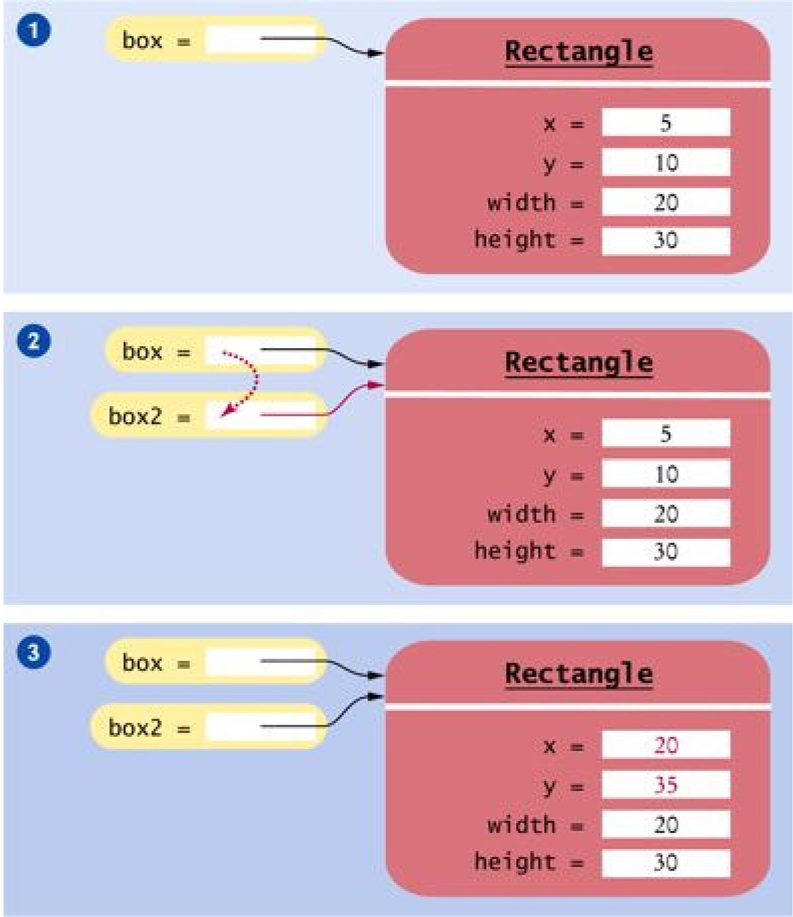
Figure 19



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

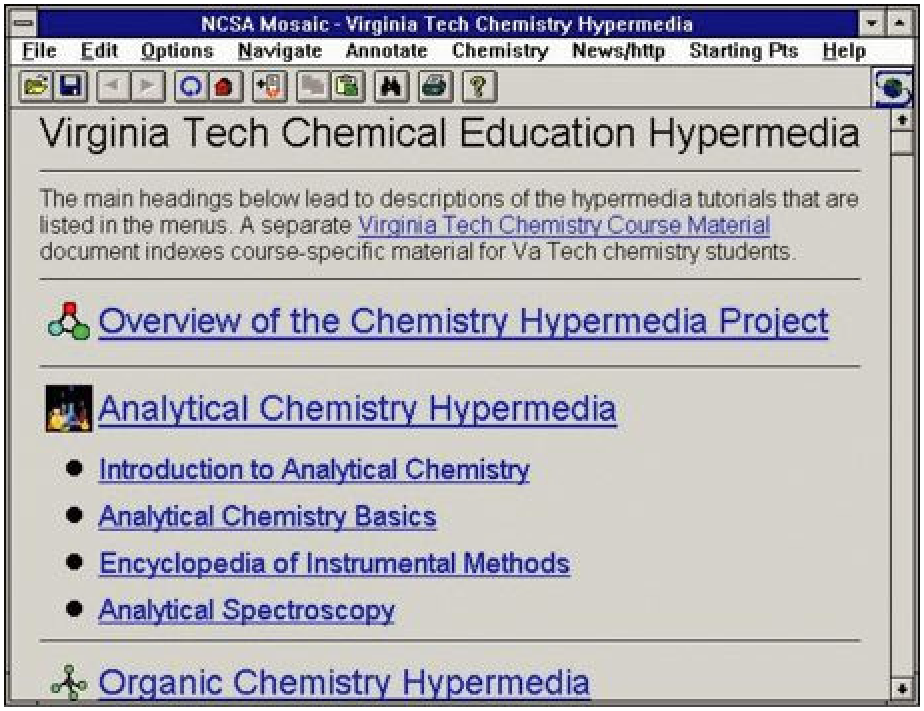
When International Business Machines Corporation (IBM), a successful manufacturer of punched-card equipment for tabulating data, first turned its attention to designing computers in the early 1950s, its planners assumed that there was a market for perhaps 50 such devices, for installation by the government, 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

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.



Virginia Tech Chemical Education Hypermedia

The main headings below lead to descriptions of the hypermedia tutorials that are listed in the menus. A separate Virginia Tech Chemistry Course Material document indexes course-specific material for Va Tech chemistry students.

Overview of the Chemistry Hypermedia Project

Analytical Chemistry Hypermedia

- Introduction to Analytical Chemistry
- Analytical Chemistry Basics
- Encyclopedia of Instrumental Methods
- Analytical Spectroscopy

Organic Chemistry Hypermedia

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. 71

72

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.

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.

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. `“Missississi”`

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));
```