

CHAPTER 2

USING OBJECTS

CHAPTER GOALS

- To learn about variables
- To understand the concepts of classes and objects
- To be able to call methods
- To learn about arguments and return values
- To be able to browse the API documentation
- To implement test programs
- To understand the difference between objects and object references
- To write programs that display simple shapes



© Lisa F. Young/iStockphoto.

CHAPTER CONTENTS

2.1 OBJECTS AND CLASSES 32

2.2 VARIABLES 34

SYN Variable Declaration 35

SYN Assignment 39

CE1 Using Undeclared or Uninitialized Variables 40

CE2 Confusing Variable Declarations and Assignment Statements 40

PT1 Choose Descriptive Variable Names 41

2.3 CALLING METHODS 41

PT2 Learn By Trying 45

2.4 CONSTRUCTING OBJECTS 46

SYN Object Construction 47

CE3 Trying to Invoke a Constructor Like a Method 48

2.5 ACCESSOR AND MUTATOR METHODS 48

2.6 THE API DOCUMENTATION 50


SYN Importing a Class from a Package 52

PT3 Don't Memorize—Use Online Help 53

2.7 IMPLEMENTING A TEST PROGRAM 53

ST1 Testing Classes in an Interactive Environment 54

WE1 How Many Days Have You Been Alive? 

WE2 Working with Pictures 

2.8 OBJECT REFERENCES 55

C&S Computer Monopoly 58

2.9 GRAPHICAL APPLICATIONS 59

2.10 ELLIPSES, LINES, TEXT, AND COLOR 64



© Lisa F. Young/iStockphoto.

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

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

2.1 Objects and Classes

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

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

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



© Luc Meaille/iStockphoto.

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

2.1.1 Using Objects

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

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

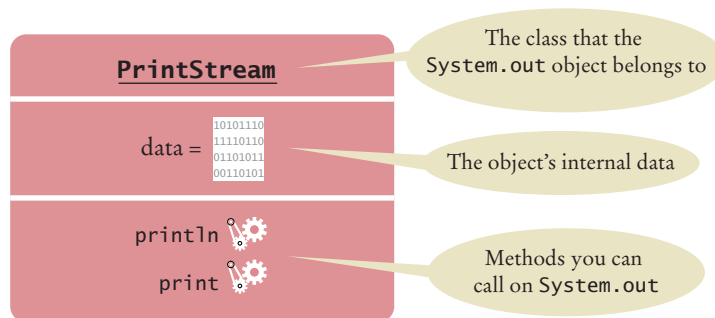


Figure 1 Representation of the `System.out` Object

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

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

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

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

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



© Steven Frame/iStockphoto.

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

2.1.2 Classes

In Chapter 1, you encountered two objects:

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

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

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

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

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

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



© Arcad Images/Alamy Inc.

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

SELF CHECK



1. In Java, objects are grouped into classes according to their behavior. Would a window object and a water heater object belong to the same class or to different classes? Why?
2. Some light bulbs use a glowing filament, others use a fluorescent gas. If you consider a light bulb a Java object with an “illuminate” method, would you need to know which kind of bulb it is?
3. What actually happens when you try to call the following?
`"Hello, World".println(System.out)`

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

2.2 Variables

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

2.2.1 Variable Declarations

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

```
int width = 20;
```

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



Javier Larrea/AGE Fotostock.

Syntax 2.1 Variable Declaration

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

The type specifies what can be done with values stored in this variable.

String greeting = "Hello, Dave!";

See page 37 for rules and examples of valid names.

A variable declaration ends with a semicolon.

Use a descriptive variable name.
 See page 41.

Supplying an initial value is optional, but it is usually a good idea.

A variable is a storage location with a name.

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

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

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

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

```
int width = 20;
```

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

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

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

Note that the type comes before the variable name:

```
int width = 20;
```



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

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

Table 1 shows several examples of variable declarations.

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



Table 1 Variable Declarations in Java	
Variable Name	Comment
<code>int width = 20;</code>	Declares an integer variable and initializes it with 20.
<code>int perimeter = 4 * width;</code>	The initial value need not be a fixed value. (Of course, <code>width</code> must have been previously declared.)
<code>String greeting = "Hi!";</code>	This variable has the type <code>String</code> and is initialized with the string <code>"Hi"</code> .
 <code>height = 30;</code>	Error: The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.2.5.
 <code>int width = "20";</code>	Error: You cannot initialize a number with the string <code>"20"</code> . (Note the quotation marks.)
<code>int width;</code>	Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 40.
<code>int width, height;</code>	Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.

2.2.2 Types

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

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

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

```
double milesPerGallon = 22.5;
```

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

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

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

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

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

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

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

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

2.2.3 Names

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

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






- 1. Names must start with a letter or the underscore (`_`) character, and the remaining characters must be letters, numbers, or underscores. (Technically, the `$` symbol is allowed as well, but you should not use it—it is intended for names that are automatically generated by tools.)
- 2. You cannot use other symbols such as `?` or `%`. Spaces are not permitted inside names either. You can use uppercase letters to denote word boundaries, as in `milesPerGallon`. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.)
- 3. Names are **case sensitive**, that is, `milesPerGallon` and `milespergallon` are different names.
- 4. You cannot use **reserved words** such as `double` or `class` as names; these words are reserved exclusively for their special Java meanings. (See Appendix C for a listing of all reserved words in Java.)



© GlobalP/Stockphoto

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

It is a convention among Java programmers that names of variables and methods start with a lowercase letter (such as `milesPerGallon`). Class names should start with an uppercase letter (such as `HelloPrinter`). That way, it is easy to tell them apart. Table 2 shows examples of legal and illegal variable names in Java.

Table 2 Variable Names in Java	
Variable Name	Comment
<code>distance_1</code>	Names consist of letters, numbers, and the underscore character.
<code>x</code>	In mathematics, you use short variable names such as <i>x</i> or <i>y</i> . This is legal in Java, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 41).
 <code>CanVolume</code>	Caution: Names are case sensitive. This variable name is different from <code>canVolume</code> , and it violates the convention that variable names should start with a lowercase letter.
 <code>6pack</code>	Error: Names cannot start with a number.
 <code>can volume</code>	Error: Names cannot contain spaces.
 <code>double</code>	Error: You cannot use a reserved word as a name.
 <code>miles/gal</code>	Error: You cannot use symbols such as <code>/</code> in names.

2.2.4 Comments

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

```
double milesPerGallon = 35.5; // The average fuel efficiency of new U.S. cars in 2013
```

This comment explains the significance of the value 35.5 to a human reader. The compiler does not process comments at all. It ignores everything from a `//` delimiter to the end of the line.

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

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

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

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

2.2.5 Assignment

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

```
int width = 10; ❶
```

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

```
width = 20; ❷
```

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

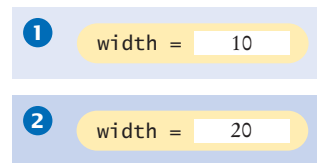


Figure 2
Assigning a New Value to a Variable

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

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

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

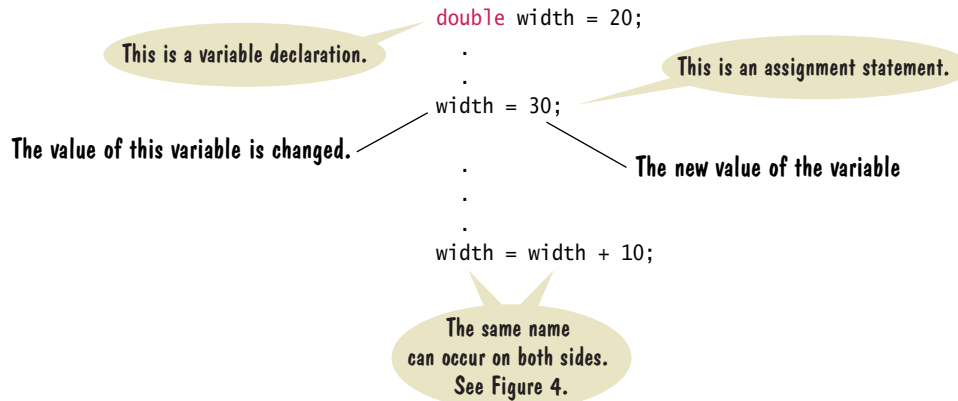
Figure 3
An Uninitialized Variable

height =

No value has been assigned.

Syntax 2.2 Assignment

Syntax *variableName = value;*



All variables must be initialized before you access them.

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

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

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

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

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

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

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

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

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

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



FULL CODE EXAMPLE

Go to wiley.com/go/bje06code to download a program that demonstrates variables and assignments.

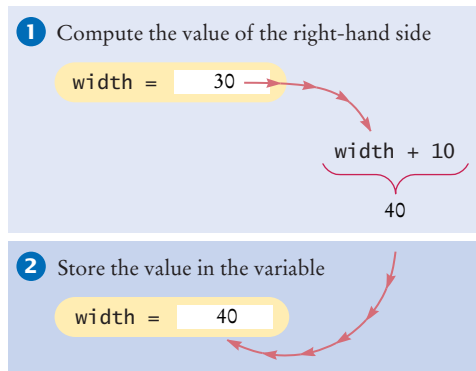


Figure 4
Executing the Statement
`width = width + 10`



4. What is wrong with the following variable declaration?
`int miles per gallon = 39.4`
5. Declare and initialize two variables, `unitPrice` and `quantity`, to contain the unit price of a single item and the number of items purchased. Use reasonable initial values.
6. Use the variables declared in Self Check 5 to display the total purchase price.
7. What are the types of the values `0` and `"0"`?
8. Which number type would you use for storing the area of a circle?
9. Which of the following are legal identifiers?
`Greeting1`
`g`
`void`
`101dalmatians`
`Hello, World`
`<greeting>`
10. Declare a variable to hold your name. Use camel case in the variable name.
11. Is `12 = 12` a valid expression in the Java language?
12. How do you change the value of the `greeting` variable to `"Hello, Nina!"`?
13. How would you explain assignment using the parking space analogy?

Practice It Now you can try these exercises at the end of the chapter: R2.4, R2.5, R2.7.

Common Error 2.1



Using Undeclared or Uninitialized Variables

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

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

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

A related error is to leave a variable uninitialized:

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

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

Common Error 2.2



Confusing Variable Declarations and Assignment Statements

Suppose your program declares a variable as follows:

```
int width = 20;
```

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

```
width = 30;
```

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

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

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

Programming Tip 2.1



Choose Descriptive Variable Names

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

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

Compare that statement with the following one:

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

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

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

2.3 Calling Methods

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

2.3.1 The Public Interface of a Class

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

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

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

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

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

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

sets `bigRiver` to the `String` object “MISSISSIPPI”.

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

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

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



© Damir Cudic/iStockphoto.

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

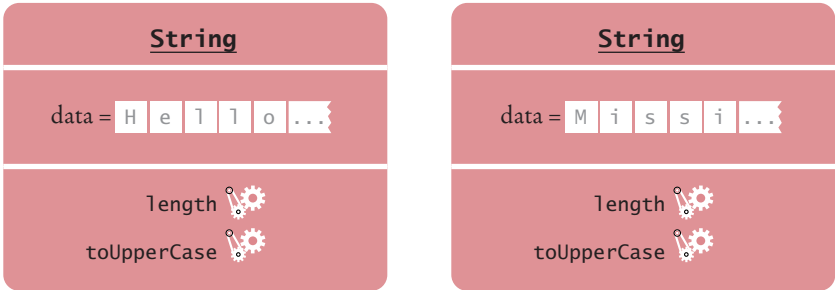


Figure 5 A Representation of Two String Objects

2.3.2 Method Arguments

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

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

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

Figure 6 illustrates passing the argument to the method.

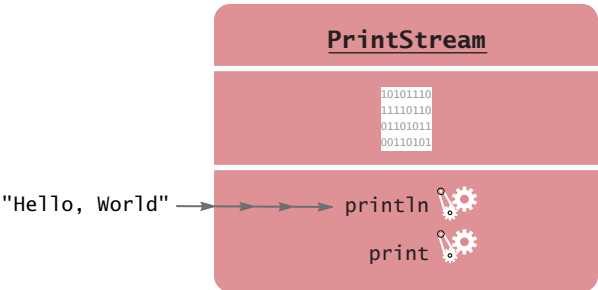


Figure 6 Passing an Argument to the println Method

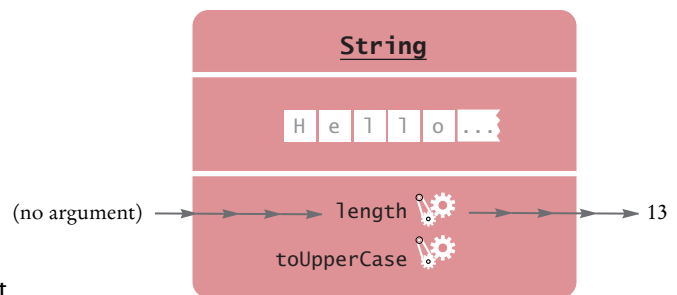
At this tailor shop, the customer's measurements and the fabric are the arguments of the sew method. The return value is the finished garment.



© Leontura/Stock photo.

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

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



2.3.3 Return Values

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

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

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

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

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

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

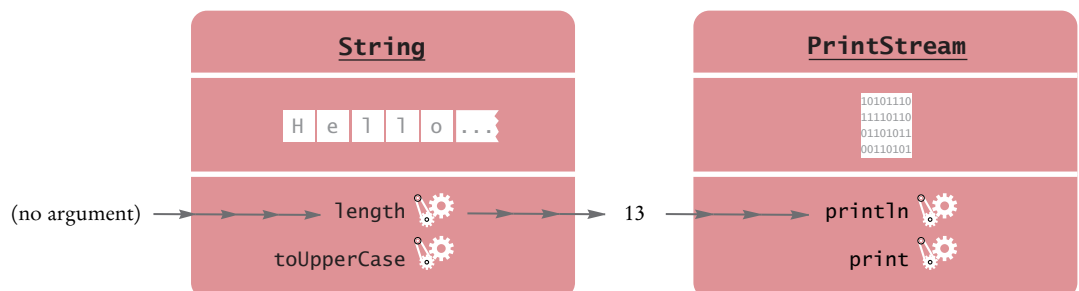


Figure 8 Passing the Result of a Method Call to Another Method

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

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

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

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

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

Or you can pass it to another method:

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

As Figure 9 shows, this method call

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

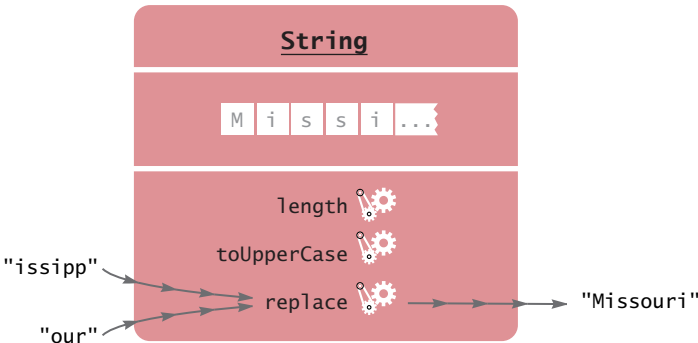


Figure 9 Calling the `replace` Method

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

2.3.4 Method Declarations

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

```
public int length()
```

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

The `replace` method is declared as

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

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

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

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

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

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

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



FULL CODE EXAMPLE

Go to wiley.com/go/bje06code to download a program that demonstrates method calls.



SELF CHECK

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

Practice It Now you can try these exercises at the end of the chapter: R2.8, R2.9, R2.10.

Programming Tip 2.2



Learn By Trying

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

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

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

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

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

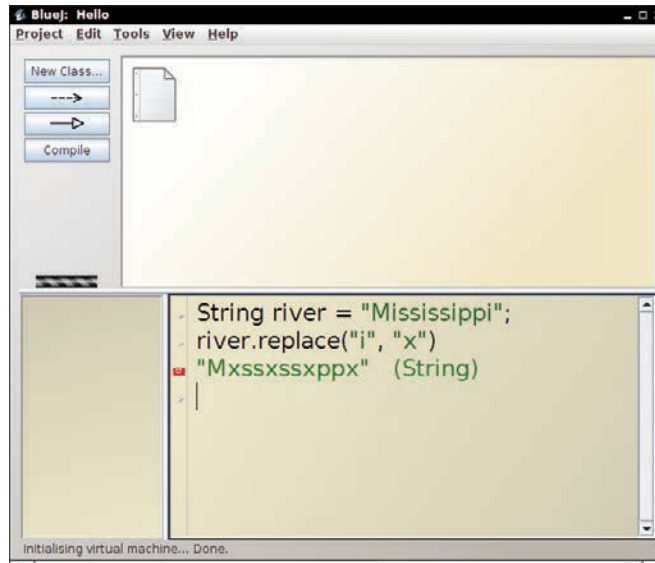


Figure 10 The Code Pad in BlueJ

2.4 Constructing Objects

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

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

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



Objects of the `Rectangle` class describe rectangular shapes.

© simankocasian/iStockphoto.

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

Figure 11 Rectangle Objects

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

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

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

Here is what happens in detail:

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

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

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

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

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

Syntax 2.3 Object Construction

Syntax `new` *ClassName*(arguments)

The `new` expression yields an object.

Construction arguments

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

Usually, you save the constructed object in a variable.

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

You can also pass a constructed object to a method.

Supply the parentheses even when there are no arguments.

**FULL CODE EXAMPLE**

Go to wiley.com/go/bjeo6code to download a program that demonstrates constructors.

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

```
new Rectangle()
```

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



21. How do you construct a square with center (100, 100) and side length 20?
22. Initialize the variables `box` and `box2` with two rectangles that touch each other.
23. The `getWidth` method returns the width of a `Rectangle` object. What does the following statement print?

```
System.out.println(new Rectangle().getWidth());
```
24. The `PrintStream` class has a constructor whose argument is the name of a file. How do you construct a `PrintStream` object with the construction argument "output.txt"?
25. Write a statement to save the object that you constructed in Self Check 24 in a variable.

Practice It Now you can try these exercises at the end of the chapter: R2.13, R2.16, R2.18.

Common Error 2.3**Trying to Invoke a Constructor Like a Method**

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

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

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

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

2.5 Accessor and Mutator Methods

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

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

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

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

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

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

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

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

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

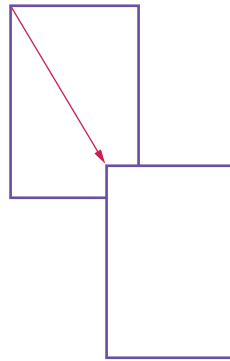


Figure 12 Using the `translate` Method to Move a Rectangle

SELF CHECK



- 26.** What does this sequence of statements print?

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

- 27.** What does this sequence of statements print?

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

- 28.** What does this sequence of statements print?

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

- 29.** Is the `toUpperCase` method of the `String` class an accessor or a mutator?

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

Practice It Now you can try these exercises at the end of the chapter: R2.19, E2.7, E2.9.

FULL CODE EXAMPLE

Go to wiley.com/go/bjeo6code to download a program that demonstrates accessors and mutators.

2.6 The API Documentation

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

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

You can find the API documentation on the Web. Point your web browser to <http://docs.oracle.com/javase/8/docs/api/index.html>. An abbreviated version of the API documentation is provided in Appendix D that may be easier to use at first, but you should eventually move on to the real thing.

2.6.1 Browsing the API Documentation

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

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

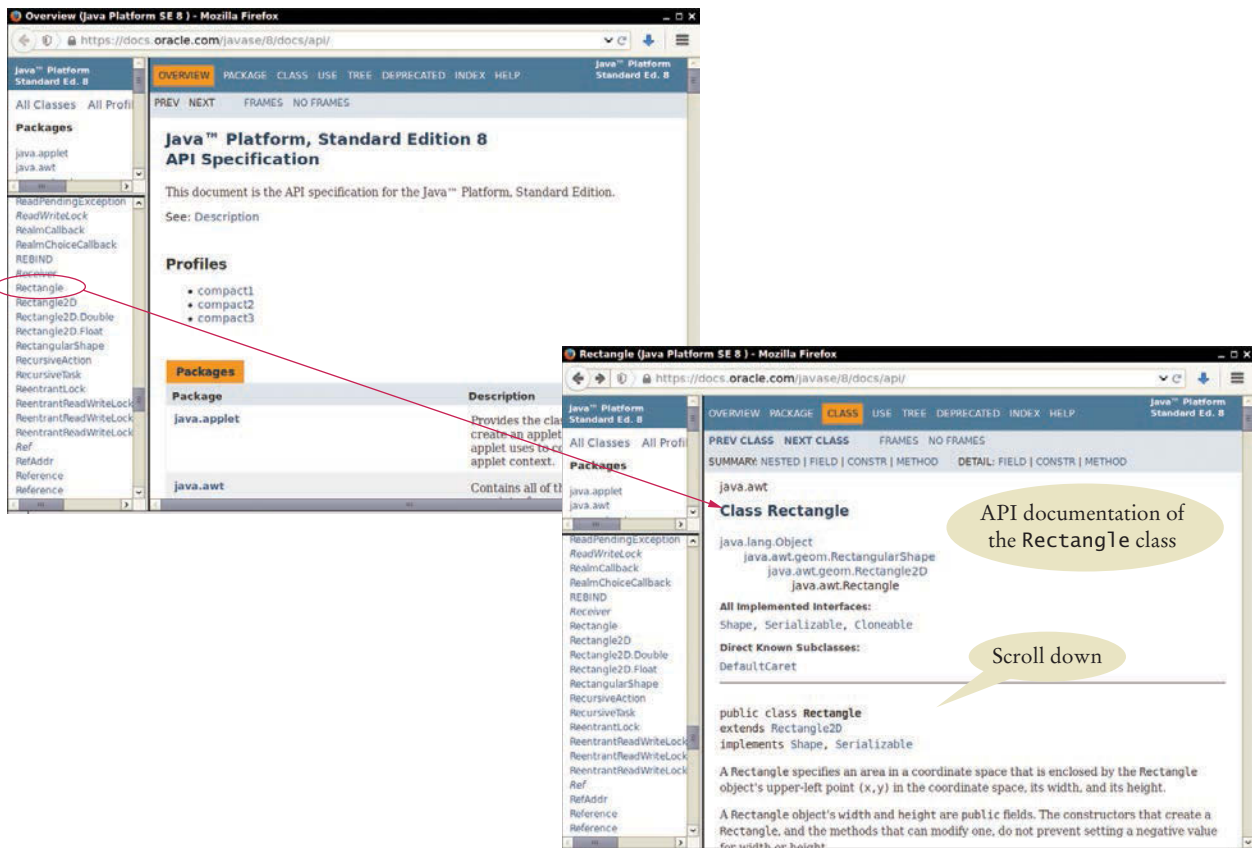


Figure 13 The API Documentation of the Standard Java Library

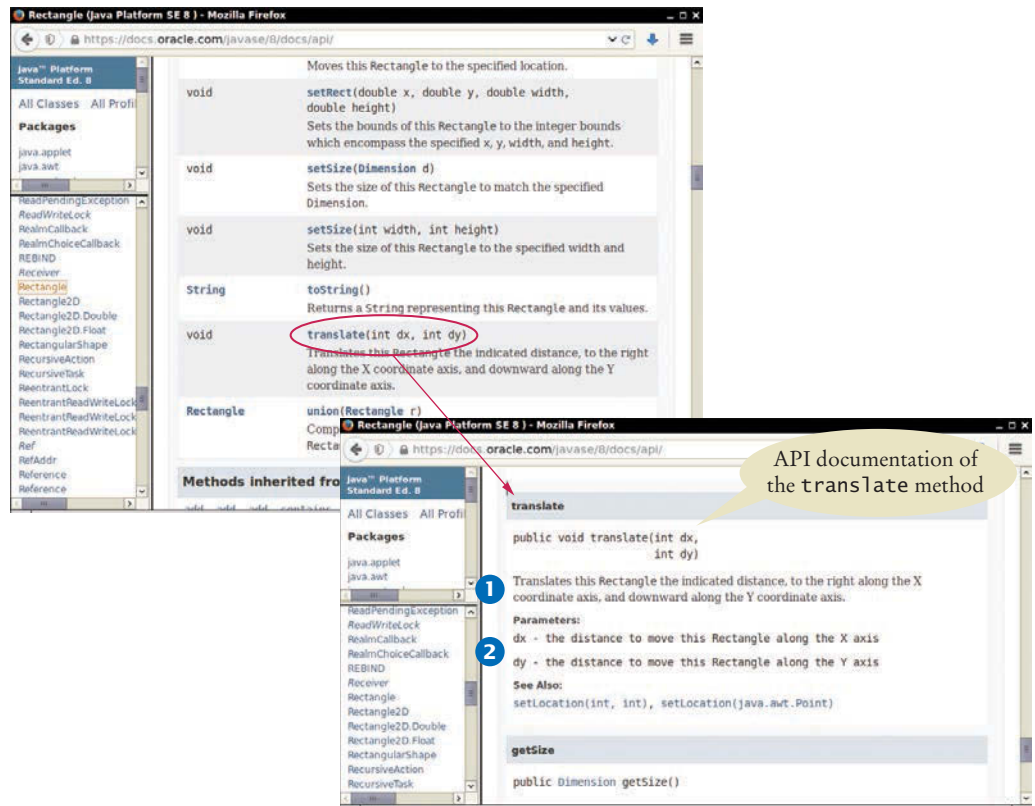


Figure 14 The Method Summary for the Rectangle Class

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

The detailed description of a method shows

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

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

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

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

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

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

2.6.2 Packages

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

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

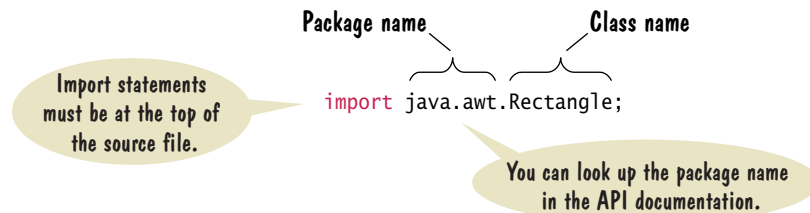
```
import java.awt.Rectangle;
```

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

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

Syntax 2.4 Importing a Class from a Package

Syntax `import packageName.ClassName;`



SELF CHECK



31. 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!"?
32. 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.)
33. Look into the API documentation of the `Rectangle` class. What is the difference between the methods `void translate(int x, int y)` and `void setLocation(int x, int y)`?
34. The `Random` class is declared in the `java.util` package. What do you need to do in order to use that class in your program?

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

Practice It Now you can try these exercises at the end of the chapter: R2.20, E2.5, E2.12.

Programming Tip 2.3



Don't Memorize—Use Online Help

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

2.7 Implementing a Test Program

A test program verifies that methods behave as expected.

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

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

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

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

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

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

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

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

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

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, so we expect an x -value of $5 + 15 = 20$ after the move. Here is the program that tests the moving of a rectangle:

section_7/MoveTester.java

```

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

Program Run

```

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

SELF CHECK



36. Suppose we had called `box.translate(25, 15)` instead of `box.translate(15, 25)`. What are the expected outputs?
37. Why doesn't the `MoveTester` program need to print the width and height of the rectangle?

Practice It Now you can try these exercises at the end of the chapter: E2.1, E2.8, E2.14.

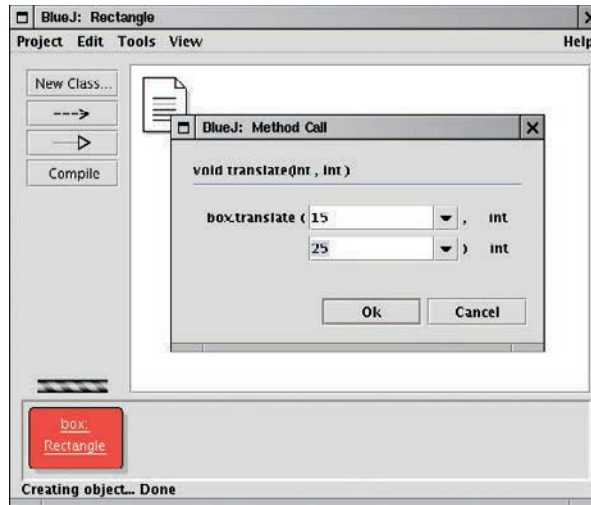
Special Topic 2.1



Testing Classes in an Interactive Environment

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

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



Testing a Method Call in BlueJ

WORKED EXAMPLE 2.1

How Many Days Have You Been Alive?



Explore the API of a class `Day` that represents a calendar day. Using that class, learn to write a program that computes how many days have elapsed since the day you were born. Go to wiley.com/go/bjeo6examples and download Worked Example 2.1.



© Constance Bannister
Corp/Hulton Archive/
Getty Images, Inc.

WORKED EXAMPLE 2.2

Working with Pictures



Learn how to use the API of a `Picture` class to edit photos. Go to wiley.com/go/bjeo6examples and download Worked Example 2.2.



Cay Horstmann.

2.8 Object References

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

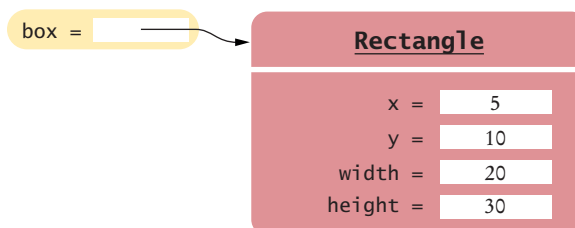


Figure 15 An Object Variable Containing an Object Reference

An object reference describes the location of an object.

© Jacob Wackerhausen/iStockphoto.



Multiple object variables can contain references to the same object.

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

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

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

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

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

```
Rectangle box2 = box;
```

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

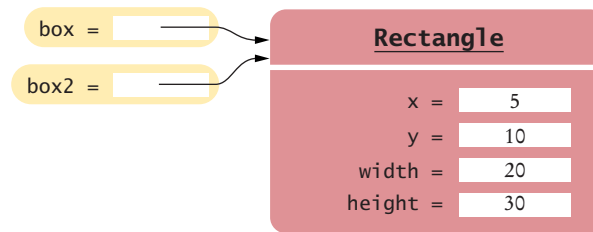


Figure 16 Two Object Variables Referring to the Same Object

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

```
int luckyNumber = 13;
```

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

`luckyNumber = 13`

Figure 17 A Number Variable Stores a Number

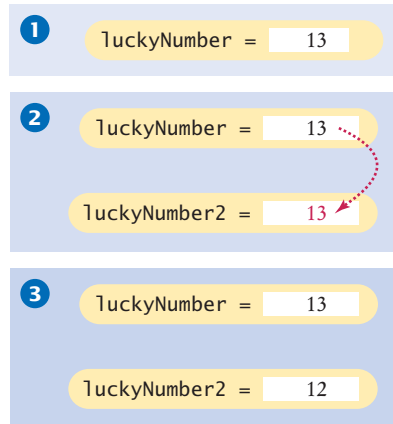
Number variables store numbers. Object variables store references.

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

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

```
int luckyNumber = 13; ❶
int luckyNumber2 = luckyNumber; ❷
luckyNumber2 = 12; ❸
```

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

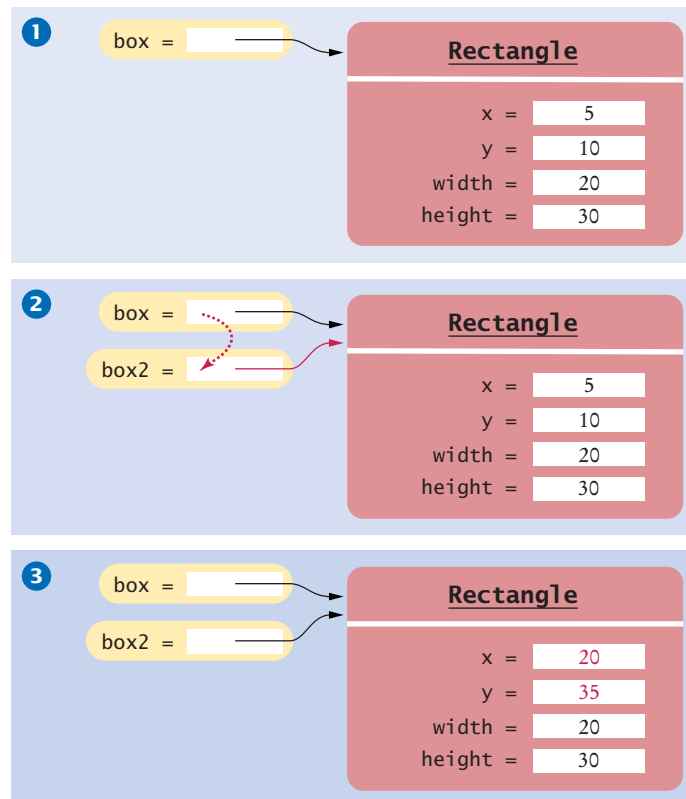
Figure 18
Copying Numbers**FULL CODE EXAMPLE**

Go to wiley.com/go/bje06code to download a program that demonstrates the difference between copying numbers and object references.

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

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

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

**Figure 19** Copying Object References

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



38. What is the effect of the assignment `String greeting2 = greeting`?
39. After calling `greeting2.toUpperCase()`, what are the contents of `greeting` and `greeting2`?

Practice It Now you can try these exercises at the end of the chapter: R2.17, R2.21.



Computing & Society 2.1 Computer Monopoly

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

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

As all of IBM's competitors fell on hard times, the U.S. government brought an antitrust suit against IBM in 1969. In the United States, it is legal to be a monopoly supplier, but it is not legal to use one's monopoly in one market to gain supremacy in another. IBM was accused of forcing customers

to buy bundles of computers, software, and peripherals, making it impossible for other vendors of software and peripherals to compete.

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

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

largely unsuccessful in establishing alternatives for desktop software.

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



A Mainframe Computer

Corbis Digital Stock.

2.9 Graphical Applications

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

2.9.1 Frame Windows

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

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



© Eduardo Jose Bernardino/iStockphoto.

A graphical application shows information inside a frame.

To show a frame, carry out the following steps:

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

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

2. Set the size of the frame:

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

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

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

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

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

4. Set the "default close operation":

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

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

5. Make the frame visible:

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

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

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

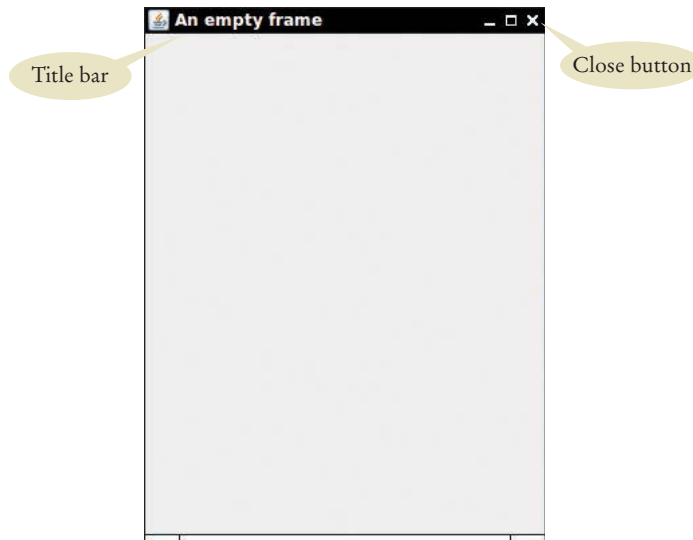


Figure 20 A Frame Window

We will go into much greater detail about Swing programming in Chapters 3, 10, and 20. For now, consider this program to be the essential plumbing that is required to show a frame.

section_9_1/EmptyFrameViewer.java

```

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

2.9.2 Drawing on a Component

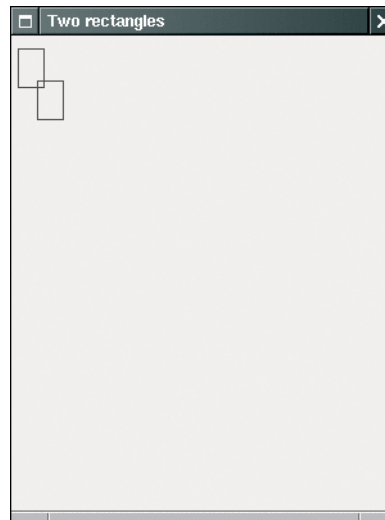
In this section, you will learn how to make shapes appear inside a frame window. The first drawing will be exceedingly modest: just two rectangles (see Figure 21). You'll soon see how to produce more interesting drawings. The purpose of this example is to show you the basic outline of a program that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

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

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

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

Figure 21
Drawing Rectangles



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

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

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

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

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

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

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

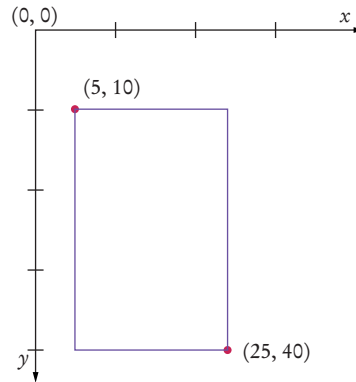
```
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // Recover Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        . . .
    }
}
```

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

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

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

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



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

section_9_2/RectangleComponent.java

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



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

2.9.3 Displaying a Component in a Frame

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

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

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

3. Add the component to the frame:

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

4. Make the frame visible.

The following listing shows the complete process.

section_9_3/RectangleViewer.java

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

Note that the rectangle drawing program consists of two classes:

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



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

Practice It Now you can try these exercises at the end of the chapter: R2.22, R2.26, E2.18.

2.10 Ellipses, Lines, Text, and Color

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



© Alexey Avdeev/iStockphoto.

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

2.10.1 Ellipses and Circles

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

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

Here is how you construct an ellipse:

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

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

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

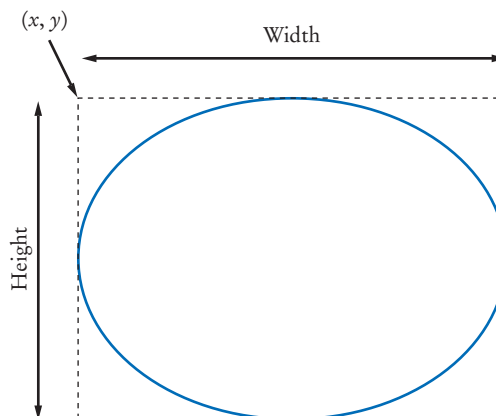


Figure 22 An Ellipse and Its Bounding Box

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

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

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

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

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

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

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

2.10.2 Lines

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

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

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

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

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

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

2.10.3 Drawing Text

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

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

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



Figure 23 Basepoint and Baseline

2.10.4 Colors

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

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

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

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

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

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

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

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

fills the inside of the circle with the current color.

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

Table 4 Predefined Colors

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



Figure 24 An Alien Face

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

section_10/FaceComponent.java

```

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

```

section_10/FaceViewer.java

```

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

```

SELF CHECK



45. Give instructions to draw a circle with center (100, 100) and radius 25.
46. Give instructions to draw a letter “V” by drawing two line segments.
47. Give instructions to draw a string consisting of the letter “V”.
48. What are the RGB color values of `Color.BLUE`?
49. How do you draw a yellow square on a red background?

Practice It Now you can try these exercises at the end of the chapter: R2.27, E2.19, E2.20.

CHAPTER SUMMARY

Identify objects, methods, and classes.

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



Write variable declarations and assignments.

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



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

Recognize arguments and return values of methods.

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



Use constructors to construct new objects.



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

Classify methods as accessor and mutator methods.

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

Use the API documentation for finding method descriptions and packages.

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

Write programs that test the behavior of methods.

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

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



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

Write programs that display frame windows.

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



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

Use the Java API for drawing simple figures.



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

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.awt.Color
java.awt.Component
    getHeight
    getWidth
    setSize
    setVisible
java.awt.Frame
    setTitle
java.awt.geom.Ellipse2D.Double
java.awt.geom.Line2D.Double
java.awt.geom.Point2D.Double
```

```
java.awt.Graphics
    setColor
java.awt.Graphics2D
    draw
    drawString
    fill
java.awt.Rectangle
    getX
    getY
    getHeight
    getWidth
```

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

REVIEW EXERCISES

- **R2.1** Explain the difference between an object and a class.
- **R2.2** Give three examples of objects that belong to the `String` class. Give an example of an object that belongs to the `PrintStream` class. Name two methods that belong to the `String` class but not the `PrintStream` class. Name a method of the `PrintStream` class that does not belong to the `String` class.
- **R2.3** What is the *public interface* of a class? How does it differ from the *implementation* of a class?
- **R2.4** Declare and initialize variables for holding the price and the description of an article that is available for sale.
- **R2.5** What is the value of `mystery` after this sequence of statements?

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

- **R2.6** What is wrong with the following sequence of statements?

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

- ■ **R2.7** Explain the difference between the = symbol in Java and in mathematics.
- ■ **R2.8** Give an example of a method that has an argument of type `int`. Give an example of a method that has a return value of type `int`. Repeat for the type `String`.
- ■ **R2.9** Write Java statements that initialize a string message with "Hello" and then change it to "HELLO". Use the `toUpperCase` method.
- ■ **R2.10** Write Java statements that initialize a string message with "Hello" and then change it to "hello". Use the `replace` method.
- ■ **R2.11** Write Java statements that initialize a string message with a message such as "Hello, world" and then remove punctuation characters from the message, using repeated calls to the `replace` method.
- **R2.12** Explain the difference between an object and an object variable.
- ■ **R2.13** Give the Java code for constructing an *object* of class `Rectangle`, and for declaring an *object variable* of class `Rectangle`.
- ■ **R2.14** Give Java code for objects with the following descriptions:
 - a. A rectangle with center (100, 100) and all side lengths equal to 50
 - b. A string with the contents "Hello, Dave"
 Create objects, not object variables.
- ■ **R2.15** Repeat Exercise R2.14, but now declare object variables that are initialized with the required objects.
- ■ **R2.16** Write a Java statement to initialize a variable `square` with a rectangle object whose top-left corner is (10, 20) and whose sides all have length 40. Then write a statement that replaces `square` with a rectangle of the same size and top-left corner (20, 20).
- ■ **R2.17** Write Java statements that initialize two variables `square1` and `square2` to refer to the same square with center (20, 20) and side length 40.
- ■ **R2.18** Find the errors in the following statements:
 - a. `Rectangle r = (5, 10, 15, 20);`
 - b. `double width = Rectangle(5, 10, 15, 20).getWidth();`
 - c. `Rectangle r;`
`r.translate(15, 25);`
 - d. `r = new Rectangle();`
`r.translate("far, far away!");`
- **R2.19** Name two accessor methods and two mutator methods of the `Rectangle` class.
- ■ **R2.20** Consult the API documentation to find methods for
 - Concatenating two strings, that is, making a string consisting of the first string, followed by the second string.
 - Removing leading and trailing white space of a string.
 - Converting a rectangle to a string.
 - Computing the smallest rectangle that contains two given rectangles.
 - Returning a random floating-point number.

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

- **R2.21** Explain the difference between an object and an object reference.
- **Graphics R2.22** What is the difference between a console application and a graphical application?
- ■ **Graphics R2.23** Who calls the `paintComponent` method of a component? When does the call to the `paintComponent` method occur?
- ■ **Graphics R2.24** Why does the argument of the `paintComponent` method have type `Graphics` and not `Graphics2D`?
- ■ **Graphics R2.25** What is the purpose of a graphics context?
- ■ **Graphics R2.26** Why are separate viewer and component classes used for graphical programs?
- **Graphics R2.27** How do you specify a text color?

PRACTICE EXERCISES

- **Testing E2.1** Write an `AreaTester` program that constructs a `Rectangle` object and then computes and prints its area. Use the `getWidth` and `getHeight` methods. Also print the expected answer.
- **Testing E2.2** Write a `PerimeterTester` program that constructs a `Rectangle` object and then computes and prints its perimeter. Use the `getWidth` and `getHeight` methods. Also print the expected answer.
- ■ **E2.3** Write a program that initializes a string with "Mississippi". Then replace all "i" with "ii" and print the length of the resulting string. In that string, replace all "ss" with "s" and print the length of the resulting string.
- **E2.4** Write a program that constructs a rectangle with area 42 and a rectangle with perimeter 42. Print the widths and heights of both rectangles.
- ■ **Testing E2.5** Look into the API documentation of the `Rectangle` class and locate the method


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

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


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

 Write a program `AddTester` that prints the expected and actual location, width, and height of `box` after the call to `add`.
- ■ **Testing E2.6** Write a program `ReplaceTester` that encodes a string by replacing all letters "i" with "!" and all letters "s" with "\$". Use the `replace` method. Demonstrate that you can correctly encode the string "Mississippi". Print both the actual and expected result.
- ■ ■ **E2.7** 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, World!"
- **Testing E2.8** The `StringBuilder` class has a method for reversing a string. In a `ReverseTester` class, construct a `StringBuilder` from a given string (such as "desserts"), call the `reverse` method followed by the `toString` method, and print the result. Also print the expected value.



- ■ **E2.9** In the Java library, a color is specified by its red, green, and blue components between 0 and 255 (see Table 4 on page 66). Write a program `BrighterDemo` that constructs a `Color` object with red, green, and blue values of 50, 100, and 150. Then apply the `brighter` method of the `Color` class and print the red, green, and blue values of the resulting color. (You won't actually see the color—see Exercise E2.10 on how to display the color.)

- ■ **Graphics E2.10** Repeat Exercise E2.9, but place your code into the following class. Then the color will be displayed.

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

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

- ■ **E2.11** Repeat Exercise E2.9, but apply the `darker` method of the `Color` class twice to the object `Color.RED`. Call your class `DarkerDemo`.
- ■ **E2.12** The `Random` class implements a *random number generator*, which produces sequences of numbers that appear to be random. To generate random integers, you construct an object of the `Random` class, and then apply the `nextInt` method. For example, the call `generator.nextInt(6)` gives you a random number between 0 and 5.
Write a program `DieSimulator` that uses the `Random` class to simulate the cast of a die, printing a random number between 1 and 6 every time that the program is run.
- ■ **E2.13** Write a program `RandomPrice` that prints a random price between \$10.00 and \$19.95 every time the program is run.
- ■ **Testing E2.14** Look at the API of the `Point` class and find out how to construct a `Point` object. In a `PointTester` program, construct two points with coordinates (3, 4) and (−3, −4). Find the distance between them, using the `distance` method. Print the distance, as well as the expected value. (Draw a sketch on graph paper to find the value you will expect.)
- **E2.15** Using the `Day` class of Worked Example 2.1, write a `DayTester` program that constructs a `Day` object representing today, adds ten days to it, and then computes the difference between that day and today. Print the difference and the expected value.
- ■ **E2.16** Using the `Picture` class of Worked Example 2.2, write a `HalfSizePicture` program that loads a picture and shows it at half the original size, centered in the window.
- ■ **E2.17** Using the `Picture` class of Worked Example 2.2, write a `DoubleSizePicture` program that loads a picture, doubles its size, and shows the center of the picture in the window.
- ■ **Graphics E2.18** Write a graphics program that draws two squares, both with the same center. Provide a class `TwoSquareViewer` and a class `TwoSquareComponent`.

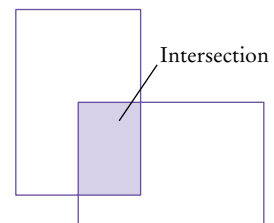
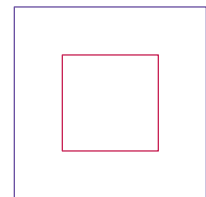
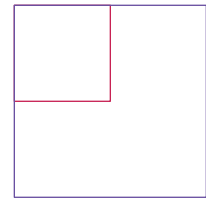
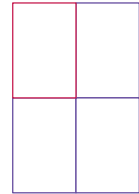
- ■ **Graphics E2.19** Write a program that draws two solid squares: one in pink and one in purple. Use a standard color for one of them and a custom color for the other. Provide a class `TwoSquareViewer` and a class `TwoSquareComponent`.
- ■ **Graphics E2.20** Write a graphics program that draws your name in red, contained inside a blue rectangle. Provide a class `NameViewer` and a class `NameComponent`.

PROGRAMMING PROJECTS

- ■ **P2.1** Write a program called `FourRectanglePrinter` that constructs a `Rectangle` object, prints its location by calling `System.out.println(box)`, and then translates and prints it three more times, so that, if the rectangles were drawn, they would form one large rectangle, as shown at right.
Your program will not produce a drawing. It will simply print the locations of the four rectangles.
- ■ **P2.2** Write a `GrowSquarePrinter` program that constructs a `Rectangle` object square representing a square with top-left corner (100, 100) and side length 50, prints its location by calling `System.out.println(square)`, applies the `translate` and `grow` methods, and calls `System.out.println(square)` again. The calls to `translate` and `grow` should modify the square so that it has twice the size and the same top-left corner as the original. If the squares were drawn, they would look like the figure at right.
Your program will not produce a drawing. It will simply print the locations of square before and after calling the mutator methods.
Look up the description of the `grow` method in the API documentation.
- ■ ■ **P2.3** Write a `CenteredSquaresPrinter` program that constructs a `Rectangle` object square representing a square with top-left corner (100, 100) and side length 200, prints its location by calling `System.out.println(square)`, applies the `grow` and `translate` methods, and calls `System.out.println(square)` again. The calls to `grow` and `translate` should modify the square so that it has half the width and is centered in the original square. If the squares were drawn, they would look like the figure at right. Your program will not produce a drawing. It will simply print the locations of square before and after calling the mutator methods.
Look up the description of the `grow` method in the API documentation.
- ■ ■ **P2.4** The intersection method computes the *intersection* of two rectangles—that is, the rectangle that would be formed by two overlapping rectangles if they were drawn, as shown at right.
You call this method as follows:

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


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



- ■ ■ **Graphics P2.5** In this exercise, you will explore a simple way of visualizing a `Rectangle` object. The `setBounds` method of the `JFrame` class moves a frame window to a given rectangle. Complete the following program to visually show the `translate` method of the `Rectangle` class:

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

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

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

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

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

© Feng Yu/Stockphoto.



- ■ ■ **P2.6** Write a program `LotteryPrinter` that picks a combination in a lottery. In this lottery, players can choose 6 numbers (possibly repeated) between 1 and 49. Construct an object of the `Random` class (see Exercise E2.12) and invoke an appropriate method to generate each number. (In a real lottery, repetitions aren't allowed, but we haven't yet discussed the programming constructs that would be required to deal with that problem.) Your program should print out a sentence such as "Play this combination—it'll make you rich!", followed by a lottery combination.
- ■ **P2.7** Using the `Day` class of Worked Example 2.1, write a program that generates a `Day` object representing February 28 of this year, and three more such objects that represent February 28 of the next three years. Advance each object by one day, and print each object. Also print the expected values:

```
2016-02-29
Expected: 2016-02-29
2017-03-01
Expected: 2017-03-01
. . .
```

- ■ ■ **P2.8** The `GregorianCalendar` class describes a point in time, as measured by the Gregorian calendar, the standard calendar that is commonly used throughout the world today. You construct a `GregorianCalendar` object from a year, month, and day of the month, like this:

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

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

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

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

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

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

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

Your task is to write a program that prints:

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

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

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



- ■ **P2.9** In Java 8, the `LocalDate` class describes a calendar date that does not depend on a location or time zone. You construct a date like this:

```
LocalDate today = LocalDate.now(); // Today's date
LocalDate eckertsBirthday = LocalDate(1919, 4, 9);
```

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

```
LocalDate later = today.plusDays(10); // Ten days from today
```

This method does not mutate the `today` object, but it returns a new object that is a given number of days away from today.

To get the year of a day, call

```
int year = today.getYear();
```

To get the weekday of a `LocalDate`, call

```
String weekday = today.getDayOfWeek().toString();
```

Your task is to write a program that prints

- The weekday of “Pi day”, that is, March 14, of the current year.
- The date and weekday of “Programmer’s day” in the current year; that is, the 256th day of the year. (The number 256, or 2^8 , is useful for some programming tasks.)
- The date and weekday of the date that is 10,000 days earlier than today.

- ■ ■ **Testing P2.10** Write a program `LineDistanceTester` that constructs a line joining the points (100, 100) and (200, 200), then constructs points (100, 200), (150, 150), and (250, 50). Print the distance from the line to each of the three points, using the `ptSegDist` method of the `Line2D` class. Also print the expected values. (Draw a sketch on graph paper to find what values you expect.)

- ■ **Graphics P2.11** Repeat Exercise P2.10, but now write a graphical application that shows the line and the points. Draw each point as a tiny circle. Use the `drawString` method to draw each distance next to the point, using calls

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

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

- ■ **Graphics P2.13** Write a program to plot the face at right. Provide a class `FaceViewer` and a class `FaceComponent`.
- ■ **Graphics P2.14** Write a graphical program that draws a traffic light.
- ■ **Graphics P2.15** Run the following program:



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

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

Modify the program as follows:

- Double the frame size.
- Change the greeting to “Hello, *your name*!”.
- Change the background color to pale green (see Exercise E2.10).
- For extra credit, add an image of yourself. (*Hint*: Construct an `ImageIcon`.)

ANSWERS TO SELF-CHECK QUESTIONS

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

15. `System.out.println(greeting.toUpperCase());`
or
`System.out.println("Hello, World!".toUpperCase());`
16. It is not legal. The variable `river` has type `String`. The `println` method is not a method of the `String` class.
17. The arguments are the strings "p" and "s".
18. "Missississii"
19. 12
20. As `public String toUpperCase()`, with no argument and return type `String`.
21. `new Rectangle(90, 90, 20, 20)`
22. `Rectangle box = new Rectangle(5, 10, 20, 30);`
`Rectangle box2 = new Rectangle(25, 10, 20, 30);`
23. 0
24. `new PrintStream("output.txt");`
25. `PrintStream out = new PrintStream("output.txt");`
26. Before: 5
After: 30
27. Before: 20
After: 20
Moving the rectangle does not affect its width or height. You can change the width and height with the `setSize` method.
28. HELLO
hello
Note that calling `toUpperCase` doesn't modify the string.
29. An accessor—it doesn't modify the original string but returns a new string with uppercase letters.
30. `box.translate(-5, -10)`, provided the method is called immediately after storing the new rectangle into `box`.
31. `toLowerCase`
32. "Hello, Space !" —only the leading and trailing spaces are trimmed.
33. The arguments of the `translate` method tell how far to move the rectangle in the *x*- and *y*-directions. The arguments of the `setLocation` method indicate the new *x*- and *y*-values for the top-left corner.
For example, `box.translate(1, 1)` moves the box one pixel down and to the right. `box.setLocation(1, 1)` moves box to the top-left corner of the screen.
34. Add the statement `import java.util.Random;` at the top of your program.
35. In the `java.math` package.
36. *x*: 30, *y*: 25
37. Because the `translate` method doesn't modify the shape of the rectangle.
38. Now `greeting` and `greeting2` both refer to the same `String` object.
39. 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.
40. Modify the `EmptyFrameViewer` program as follows:
`frame.setSize(300, 300);`
`frame.setTitle("Hello, World!");`
41. Construct two `JFrame` objects, set each of their sizes, and call `setVisible(true)` on each of them.
42. Change line 17 of `RectangleComponent` to
`Rectangle box = new Rectangle(5, 10, 20, 20);`
43. Replace the call to `box.translate(15, 25)` with
`box = new Rectangle(20, 35, 20, 20);`
44. The compiler complains that `g` doesn't have a `draw` method.
45. `g2.draw(new Ellipse2D.Double(75, 75, 50, 50));`
46. `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);`
47. `g2.drawString("V", 0, 30);`
48. 0, 0, 255
49. 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));`



WORKED EXAMPLE 2.1

How Many Days Have You Been Alive?



Many programs need to process dates such as “February 15, 2010”. The `worked_example_1` directory of this chapter’s companion code contains a `Day` class that was designed to work with calendar days.

The `Day` class knows about the intricacies of our calendar, such as the fact that January has 31 days and February has 28 or sometimes 29. The Julian calendar, instituted by Julius Caesar in the first century BCE, introduced the rule that every fourth year is a leap year. In 1582, Pope Gregory XIII ordered the implementation of the calendar that is in common use throughout the world today, called the Gregorian calendar. It refines the leap year rule by specifying that years divisible by 100 are not leap years, unless they are divisible by 400. Thus, the year 1900 was not a leap year but the year 2000 was. All of these details are handled by the internals of the `Day` class.

The `Day` class lets you answer questions such as

- How many days are there between now and the end of the year?
- What day is 100 days from now?

Problem Statement Your task is to write a program that determines how many days you have been alive. You should *not* look inside the internal implementation of the `Day` class. Use the API documentation by pointing your browser to the file `index.html` in the `ch02/worked_example_1/api` subdirectory.



© Constance Bannister Corp/Hulton Archive/ Getty Images, Inc.

As you can see from the API documentation (see figure on next page), you construct a `Day` object from a given year, month, and day, like this:

```
Day jamesGoslingsBirthday = new Day(1955, 5, 19);
```

There is a method for adding days to a given day, for example:

```
Day later = jamesGoslingsBirthday.addDays(100);
```

You can then find out what the result is, by applying the `getYear/getMonth/getDate` methods:

```
System.out.println(later.getYear());
System.out.println(later.getMonth());
System.out.println(later.getDate());
```

However, that approach does not solve our problem (unless you are willing to replace 100 with other values until, by trial and error, you obtain today’s date). Instead, use the `daysFrom` method. According to the API documentation, we need to supply another day. That is, the method is called like this:

```
int daysAlive = day1.daysFrom(day2);
```

In our situation, one of the `Day` objects is `jamesGoslingsBirthday`, and the other is today’s date. This can be obtained with the constructor that has no arguments:

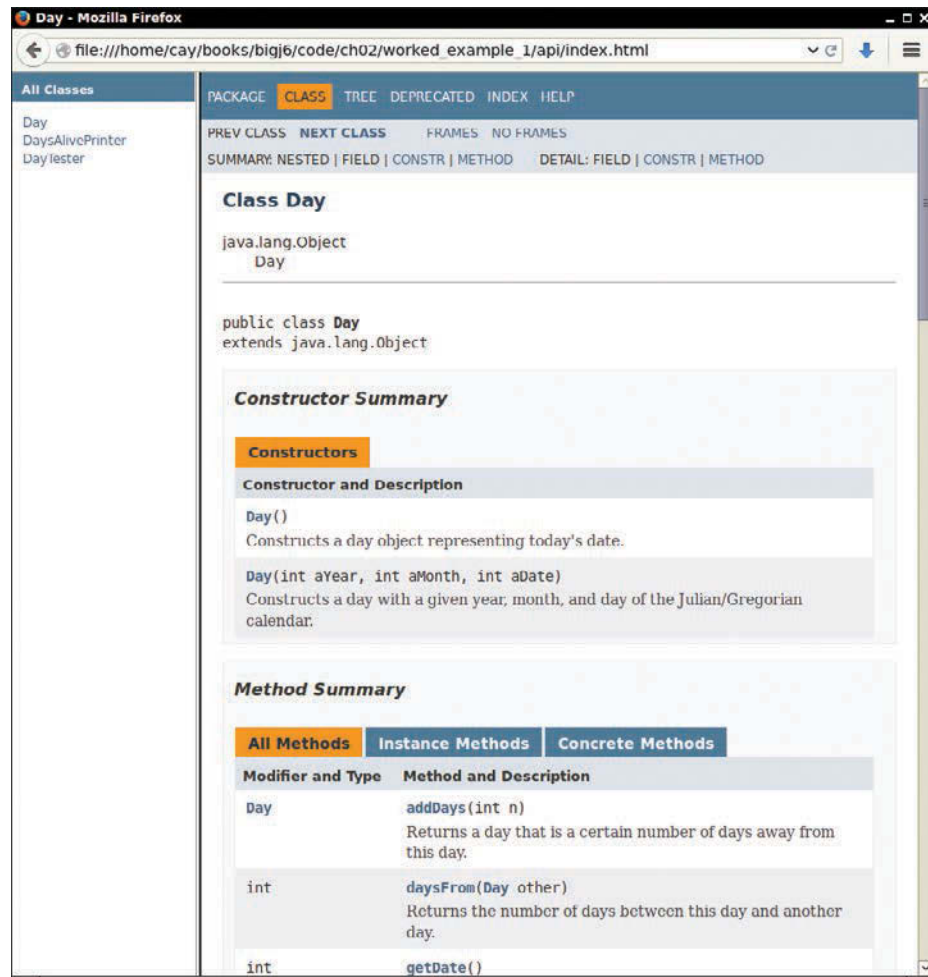
```
Day today = new Day();
```

We have two candidates on which the `daysFrom` method could be invoked, yielding the call

```
int daysAlive = jamesGoslingsBirthday.daysFrom(today);
```

or

```
int daysAlive = today.daysFrom(jamesGoslingsBirthday);
```



Which is the right choice? Fortunately, the author of the `Day` class has anticipated this question. The detail comment of the `daysFrom` method contains this statement:

Returns: the number of days that this day is away from the other
(larger than 0 if this day comes later than other)

We want a positive result. Therefore, the second form is the correct one.

Here is the program that solves our problem (see `ch02/worked_example_1` in your source code):

worked_example_1/DaysAlivePrinter.java

```

1 public class DaysAlivePrinter
2 {
3     public static void main(String[] args)
4     {
5         Day jamesGoslingsBirthday = new Day(1955, 5, 19);
6         Day today = new Day();
7         System.out.print("Today: ");
8         System.out.println(today.toString());
9         int daysAlive = today.daysFrom(jamesGoslingsBirthday);

```

```
10      System.out.print("Days alive: ");  
11      System.out.println(daysAlive);  
12  }  
13 }
```

Program Run

```
Today: 2015-02-09  
Days alive: 21826
```



WORKED EXAMPLE 2.2

Working with Pictures



Problem Statement Edit and display image files in the `Picture` class found in the `ch02/worked_example_2` directory of this chapter's companion code.

For example, the following program simply shows the image given below:

```
public class PictureDemo
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
    }
}
```



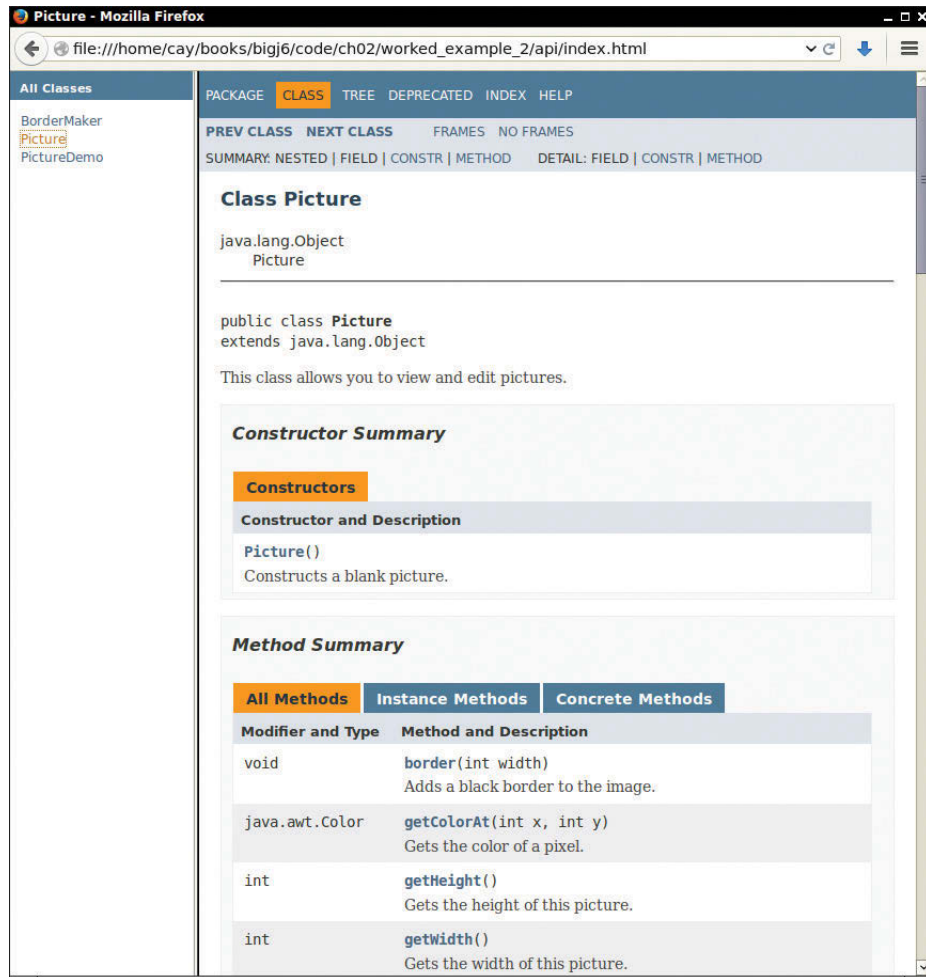
Cay Horstmann.

Your task is to write a program that reads in an image, shrinks it, and adds a border. Shrink it sufficiently so that there is a transparent border inside the black border, as in the figure below.



Cay Horstmann.

You should *not* look inside the internal implementation of the `Picture` class. Instead, use the API documentation by pointing your browser to the file `index.html` in the `worked_example_2/picture/api` subdirectory.



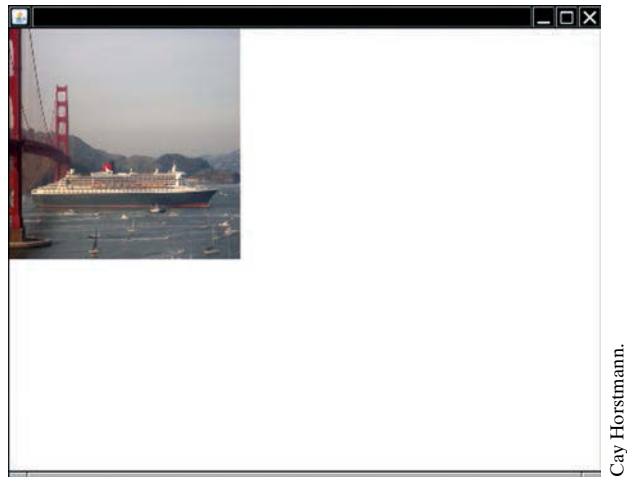
The API contains a number of methods that are unrelated to the task, but two of the methods are clearly useful:

```
public void scale(int newWidth, int newHeight)
public void border(int width)
```

If the method comments are not clear, it is a good idea to write a couple of simple test programs to see their effect. For example, this program demonstrates the `scale` method:

```
public class PictureScaleDemo
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
        pic.scale(200, 200);
    }
}
```

Here is the result:



Cay Horstmann.

As you can see, the picture has been resized to a 200×200 pixel square.

That's not quite what we want. We want the picture to be a bit smaller than the original. Let's say that the black border is 10 pixels thick, and we want another transparent border of 10 pixels. Then the target width and height are 40 pixels less than the original, leaving 20 pixels on each side for the borders.

Looking at the API, we find methods for obtaining the original width and height. Therefore, we will call

```
int newWidth = pic.getWidth() - 40;
int newHeight = pic.getHeight() - 40;
pic.scale(newWidth, newHeight);
```

Then we add the border:

```
pic.border(10);
```

The result is



Cay Horstmann.

If we can move the picture a bit before applying the border, we are done. Another look at the API reveals a method

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

That's just what we need. The picture needs to be moved 20 pixels down and to the right. Our final program is

worked_example_2/BorderMaker.java

```
1 public class BorderMaker
2 {
3     public static void main(String[] args)
4     {
5         Picture pic = new Picture();
6         pic.load("queen-mary.png");
7         int newWidth = pic.getWidth() - 40;
8         int newHeight = pic.getHeight() - 40;
9         pic.scale(newWidth, newHeight);
10        pic.move(20, 20);
11        pic.border(10);
12    }
13 }
```

Couldn't we have achieved the same result with an image editing program such as Photoshop or GIMP? Yes, but it is an easy matter to extend this program so that it can automatically apply a border to any number of images.
