CHAPTER 3

Object-Oriented Programming, Part 1: Using Classes

CHAPTER CONTENTS

Introduction

Writing computer programs that use classes and objects is called **object-oriented programming**, or **OOP**. Every Java program consists of at least one class.

In this chapter, we'll introduce object-oriented programming as a way to use classes that have already been written. Classes provide services to the program. These services might include writing a message to the program's user, popping up a dialog box, performing some mathematical calculations, formatting numbers, drawing shapes in a window, or many other basic tasks that add a more professional look to even simple programs. The program that uses a class is called the **client** of the class.

One benefit of using a prewritten class is that we don't need to write the code ourselves; it has already been written and tested for us. This means that we can write our programs more quickly. In other words, we shorten the development time of the program. Using prewritten and pretested classes provides other benefits as well, including more reliable programs with fewer errors.

In Chapter 7, we'll show you how to write your own classes. For now, we'll explore how using prewritten classes can add functionality to our programs.

3.1 Class Basics and Benefits

In Java, classes are composed of data and operations—or functions—that operate on the data. Objects of a class are created using the class as a template, or guide. Think of the class as a generic description, and an object as a specific item of that class. Or you can think of a class as a cookie cutter, the objects of that class are the cookies made with the cookie cutter. For example, a *Student* class might have the following data: name, year, and grade point average. All students have these three data items. We can create an object of the *Student* class by specifying an identifier for the object (for example, *student1*) along with a name, year, and grade point average for a particular student (for example, *Maria Gonzales*, *Sophomore*, *3.5*). The identifier of the object is called the **object reference**. Creating an object of a class is called **instantiating an object**, and the object is called an **instance of the class**. Many objects can be instantiated from one class. There can be

many instances of the *Student* class, that is, many *Student* objects can be instantiated from the *Student* class. For example, we could create a second object of the *Student* class, *student2*, with its data as *Mike Smith*, *Junior*, *3.0*.

The data associated with an object of a class are called **instance variables**, or **fields**, and can be variables and constants of any primitive data type (*byte*, *short*, *int*, *long*, *float*, *double*, *char*, and *boolean*), or they can be objects of other classes.

The operations for a class, called **methods**, set the values of the data, retrieve the current values of the data, and perform other class-related functions on the data. For example, the *Student* class would provide methods to set the values of the name, year, and grade point average; retrieve the current values of the name, year, and grade point average; and perhaps promote a student to the next year. Invoking a method on an object is called **calling the method**. With a few exceptions, only class methods can directly access or change the instance variables of an object. Other objects must call the methods to set or retrieve the values of the instance variables. Together, the fields and methods of a class are called its **members**.

In essence, a class is a new data type, which is created by combining items of Java primitive data types and objects of other classes. Just as the primitive data types can be manipulated using arithmetic operators (+, –, *, /, and %), objects can be manipulated by calling class methods.

We like to think of classes as similar to M&M™ candies: a protective outer coating around a soft center. Because the methods to operate on the data are included in the class, they provide a protective coating around the data inside. In a well-designed class, only the class methods can change the data. Methods of other classes cannot directly access the data. We say that the data is *private* to the class. In other words, the class **encapsulates** the data and the methods provide the only interface for setting or changing the data values. The benefit from this encapsulation is that the class methods ensure that only valid values are assigned to an object. For example, a method to set a Student's grade point average would accept values only between 0.0 and 4.0.

Let's look at another example of a class. The *SimpleDate* class, written by the authors, has the instance variables *month*, *day*, and *year*. An object of this class, *independenceDay*, could be instantiated with data values of *7*, *4*, and *1776*. Another object of that class, *examDay*, might be instantiated with

the values *12*, *4*, and *2006*. Methods of the *SimpleDate* class ensure that only valid values are set for the month, day, and year. For example, the class methods would not allow us to set a date with a value of January 32. Other class methods increment the date to the next day and provide the date in *mm/dd/yyyy* format.

SOFTWARE ENGINEERING TIP

By convention, class names in Java start with a capital letter. Method names, instance variables, and object names start with a lowercase letter. In all of these names, embedded words begin with a capital letter.

Notice that the class names we used, *Student* and *SimpleDate*, begin with a capital letter, and the object names, *student1*, *independenceDay*, and *examDay*, start with a lowercase letter. By convention, class names start with a capital letter. Object names, instance variables, and method names conventionally start with a lowercase letter. Internal words start with a capital letter in class names, object names, variables, and methods.

There are many benefits to using classes in a program. Some of the most important benefits include reusability (not only in the current program but also in other programs), encapsulation, and reliability.

A well-written class can be reused in many programs. For example, a *SimpleDate* class could be used in a calendar program, an appointment-scheduling program, an online shopping program, and many more applications that rely on dates. Reusing code is much faster than writing and testing new code. As an added bonus, reusing a tested and debugged class in another program makes the program more reliable.

Encapsulation of a class's data and methods helps to isolate operations on the data. This makes it easier to track the source of a bug. For example, when a bug is discovered in an object of the *Student* class, then you know to look for the problem in the methods of the *Student* class, because no other code in your program can directly change the data in a *Student* object.

You do not need to know the implementation details of a class in order to use it in your program. Does the *SimpleDate* class store the date in memory as three integers, *month*, *day*, and *year*? Or is the date stored as the number of milliseconds since 1980? The beauty of object orientation is that we don't need to know the implementation of the class; all we need to know is the class **application programming interface (API)**, that is, how to instantiate objects and how to call the class methods.

The benefits of using classes are clear. We will leave the details of creating our own classes until Chapter 7. In the meantime, let's explore how to use classes that are already written.

3.2 Creating Objects Using Constructors

A class describes a generic template for creating, or instantiating, objects. In fact, an object must be instantiated before it can be used. To understand how to instantiate an object of a class and how to call methods of the class, you must know the API of a class, which the creators of the class make public. Table 3.1 shows the API of the *SimpleDate* class, written by the authors of this textbook.

Instantiating an object consists of defining an object reference—which will hold the address of the object in memory—and calling a special method of the class called a **constructor**, which has the same name as the class. The job of the constructor is to assign initial values to the data of the class.

Example 3.1 illustrates how to instantiate objects of the *SimpleDate* class.

```
1 /*  A Demonstration of Using Constructors
2     Anderson, Franceschi
3 */
4
5 public class Constructors
6 {
7    public static void main( String [ ] args )
8    {
9      SimpleDate independenceDay;
10     independenceDay = new SimpleDate( 7, 4, 1776 );
11
12     SimpleDate graduationDate = new SimpleDate( 5, 15, 2012 );
13
14     SimpleDate defaultDate = new SimpleDate( );
15   }
16 }
```

EXAMPLE 3.1 Demonstrating Constructors

Declaring an object reference is very much like declaring a variable of a primitive type; you specify the data type and an identifier. For example, to declare an integer variable named *number1*, you provide the data type (*int*) and the identifier (*number1*), as follows:

```
int number1;
```

TABLE 3.1 The *SimpleDate* Class API

SimpleDate Class Constructor Summary

`SimpleDate( )`

creates a *SimpleDate* object with initial default values of 1, 1, 2000.

`SimpleDate( int mm, int dd, int yy )`

creates a *SimpleDate* object with the initial values of *mm*, *dd*, and *yy*.

SimpleDate Class Method Summary

Return value	Method name and argument list
`int`	`getMonth( )` returns the value of *month*.
`int`	`getDay( )` returns the value of *day*.
`int`	`getYear( )` returns the value of *year*.
`void`	`setMonth( int mm )` sets the *month* to *mm*; if *mm* is invalid, sets *month* to 1.
`void`	`setDay( int dd )` sets the *day* to *dd*; if *dd* is invalid, sets *day* to 1.
`void`	`setYear( int yy )` sets the *year* to *yy*.
`void`	`nextDay( )` increments the date to the next day.
`String`	`toString( )` returns the value of the date in the form: *month*/*day*/*year*.
`boolean`	`equals( Object obj )` compares this *SimpleDate* object to another *SimpleDate* object.

One notable difference in declaring an object reference is that its data type is a class, not a primitive data type. Here is the syntax for declaring an object reference:

```
ClassName objectReference1, objectReference2, ...;
```

In Example 3.1, lines 9, 12, and 14 declare object references for a *SimpleDate* object. *SimpleDate*, the class name, is the data type, and *independenceDay*, *graduationDate*, and *defaultDate* are the object references.

Object references can refer to **any** object of its class. For example, *SimpleDate* object references can point to any *SimpleDate* object, but a *SimpleDate* object reference cannot point to objects of other classes, such as a *Student* object.

Once an object reference has been declared, you instantiate the object using the following syntax:

```
objectReference = new ClassName( argument list );
```

This calls a constructor of the class to initialize the data. The **argument list** consists of a comma-separated list of initial data values to assign to the object. Classes often provide multiple constructors with different argument lists. Depending on which constructor you call, you can accept default values for the data or specify initial values for the data. When you instantiate an object, your argument list—that is, the number of arguments and their data types—must match one of the constructors' argument lists.

As shown in Table 3.1, the *SimpleDate* class has two constructors. The first constructor, *SimpleDate( )*, is called the **default constructor**, because its **argument list is empty**. This constructor assigns default values to all data in the object. Thus, in line 14 of Example 3.1, which uses the default constructor, the data for the *defaultDate* object is set to the default values for the *SimpleDate* class, which are *1*, *1*, and *2000*.

We see from Table 3.1 that the second constructor for the *SimpleDate* class, *SimpleDate(int mm, int dd, int yy)*, takes three arguments, all of which should evaluate to integer values. The first argument is the value for the month, the second argument is the value for the day, and the third argument is the value for the year.

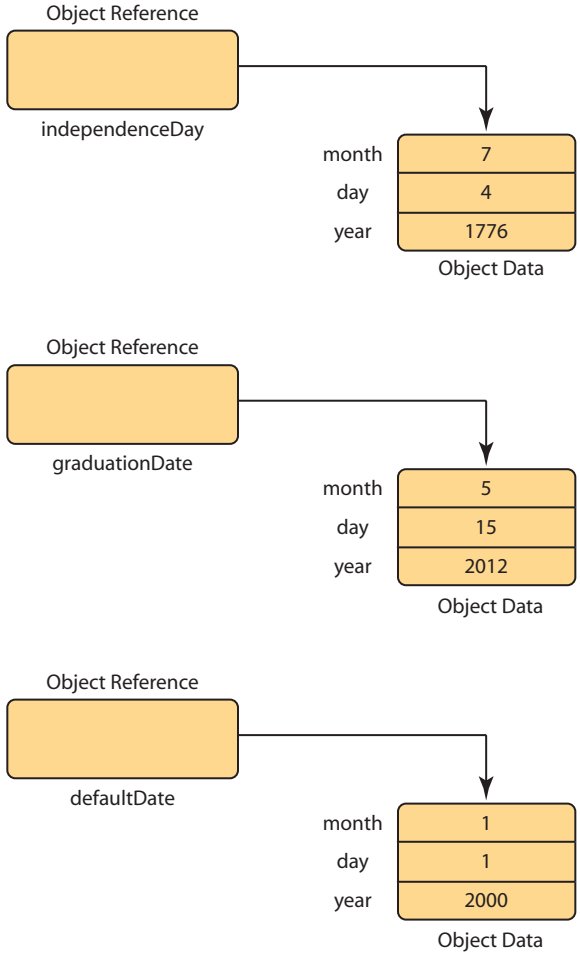
Lines 10 and 12 of Example 3.1 instantiate *SimpleDate* objects using the second constructor. In line 10, the argument list tells the constructor to give the value *7* to the month, *4* to the day, and *1776* to the year. In line 12, the

argument list tells the constructor to give the value *5* to the month, *15* to the day, and *2012* to the year. Note that no data types are given in the argument list, only the initial values for the data. The data types of the arguments are specified in the API so that the client of the class knows what data types the constructor is expecting for its arguments.

Lines 12 and 14 also illustrate that you can combine the declaration of the object reference and instantiation of the object in a single statement.

When an object is instantiated, the JVM allocates memory to the new object and assigns that memory location to its object reference. Figure 3.1 shows the three objects instantiated in Example 3.1.

Figure 3.1
Three *SimpleDate* Objects after Instantiation



It's important to understand that an object reference and the object data are different: The object reference represents the memory location, and the object data are the data stored at that memory location. Notice in Figure 3.1 that the object references, *independenceDay*, *graduationDate*, and *defaultDate*, point to the locations of the object data.

COMMON ERROR TRAP

Do not forget to instantiate all objects that your program needs. Objects must be instantiated before they can be used.

3.3 Calling Methods

Once an object is instantiated, we can use the object by calling its methods. As we mentioned earlier, the authors of classes publish their API so that their clients know what methods are available and how to call those methods.

Figure 3.2 illustrates how calling a class method alters the flow of control in your program. When this program starts running, the JVM executes instruction 1, then instruction 2, then it encounters a method call. At that

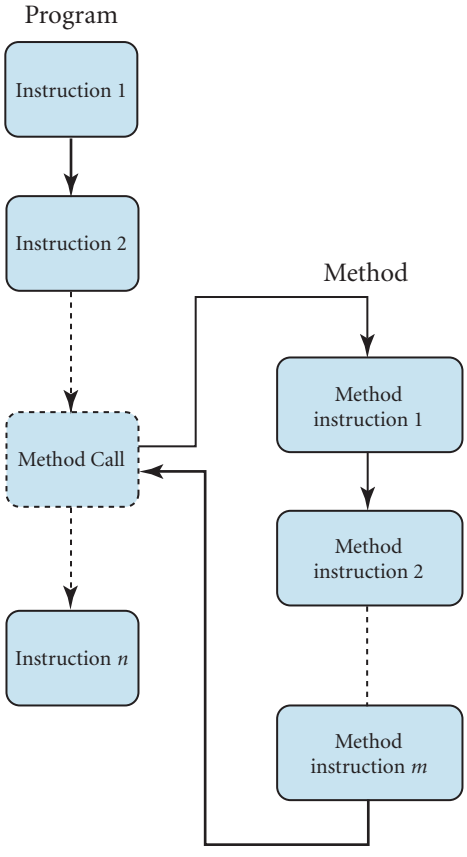


Figure 3.2
Flow of Control of a Method Call

point, the JVM **transfers control to the method** and starts executing instructions in the method. When the method finishes executing, the JVM transfers control back to the program immediately after the point the method was called and continues executing instructions in the program.

A class API consists of the class method names, their return values, and their argument lists. The argument list for a method indicates the order and number of arguments to send to the method, along with the data type of each argument. Each item in the argument list consists of a data type and a name. The arguments can be literals, constants, variables, or any expression that evaluates to the data type specified in the API of the method. For example, the API in Table 3.1 shows that the *setMonth* method takes one argument, which must evaluate to an integer value.

A method may or may not return a value, as indicated by a data type, class type, or the keyword **void** in front of the method name. If the method returns a value, then the data type or class type of its **return value** will precede the method's name. For instance, in Table 3.1, the *getDay* method returns an integer value. The call to a **value-returning method** will be used in an expression. When the method finishes executing, its return value will replace the method call in the expression. If the keyword *void* precedes the method name, the method does not return a value. Because methods with a *void* return type have no value, they cannot be used in an expression; instead, a method call to a method with a *void* return type is a complete statement. In Table 3.1, the *setYear* method is a *void* method.

Another keyword you will see preceding the method call in an API is *public*. This keyword means that any client of the class can call this method. If the keyword *private* precedes the method name, only other methods of that class can call that method. Although we will not formally include the *public* keyword in the API, all the methods we discuss in this chapter are *public*.

To call a method for an object of a class, we use **dot notation**, as follows:

```
objectReference.methodName( arg1, arg2, arg3, . . . )
```

The object reference is followed immediately by a **dot** (a period), which is followed immediately by the method name. (Later in the chapter, when we call *static* methods, we will substitute the class name for the object reference.) The arguments for the method are enclosed in parentheses.

Let's look again at the methods of the *SimpleDate* class. The first three methods in the *SimpleDate* class API take an empty argument list and return an *int*; thus, those methods have a return value of type *int*. You can

call these methods in any expression in your program where you could use an *int*. The value of the first method, *getMonth( )*, is the value of the month in the object. Similarly, the value of *getDay( )* is the value of the day in the object, and the value of *getYear( )* is the value of the year. These "get" methods are formally called **accessor methods**; they enable clients to access the value of the instance variables of an object.

The next three methods in the *SimpleDate* class API take one argument of type *int* and do not return a value, which is indicated by the keyword *void*. These methods are called in standalone statements. The first method, *setMonth(int mm)*, changes the value of the month in the object to the value of the method's argument, *mm*. Similarly, *setDay(int dd)* changes the value of the day in the object, and *setYear(int yy)* changes the value of the year in the object to the value of the method's argument. These "set" methods are formally called **mutator methods**; they enable a client to change the value of the instance variables of an object.

Example 3.2 illustrates how to use some of the methods of the *SimpleDate* class. Line 10 calls the *getMonth* method for the *independenceDay* object. When line 10 is executed, control transfers to the *getMonth* method. When the *getMonth* method finishes executing, the value it returns (7) replaces the method call in the statement. The statement then effectively becomes:

```
int independenceMonth = 7;
```

In lines 15–16, we print the value of the day in the *graduationDate* object. Again, control transfers to the *getDay* method, then its return value (15) replaces the method call. So the statement effectively becomes:

```
System.out.println( "The current day for graduation is "
                    + 15 );
```

Line 18 calls the *setDay* method, which is used to change the value of the day for an object. The *setDay* method takes one *int* argument and has a *void* return value. Line 18 is a complete statement, because the method call to a method with a *void* return value is a complete statement. The method changes the value of the day in the *graduationDate* object, which we illustrate in lines 19–20 by printing the new value as shown in Figure 3.3. Then, on line 22, we instantiate another object, *currentDay*, with a day, month, and year of 9, 30, 2008, which we demonstrate by printing the values returned by calls to the *getDay*, *getMonth*, and *getYear* methods. On line 28, we call the *nextDay* method, which has a *void* return value, and increments the date to the next day, and then we print the new values of the *currentDay* object.

COMMON ERROR TRAP

When calling a method that takes no arguments, remember to include the empty parentheses after the method's name. The parentheses are required even if there are no arguments.

COMMON ERROR TRAP

When calling a method, include only values or expressions in your argument list. Including data types in your argument list will cause a compiler error.

```
1 /*  A demonstration of calling methods
2     Anderson, Franceschi
3 */
4
5 public class Methods
6 {
7   public static void main( String [ ] args )
8   {
9      SimpleDate independenceDay = new SimpleDate( 7, 4, 1776 );
10     int independenceMonth = independenceDay.getMonth( );
11     System.out.println( "Independence day is in month "
12                         + independenceMonth );
13
14     SimpleDate graduationDate = new SimpleDate( 5, 15, 2008 );
15     System.out.println( "The current day for graduation is "
16                         + graduationDate.getDay( ) );
17
18     graduationDate.setDay( 12 );
19     System.out.println( "The revised day for graduation is "
20                         + graduationDate.getDay( ) );
21
22     SimpleDate currentDay = new SimpleDate( 9, 30, 2008 );
23     System.out.println( "The current day is "
24                         + currentDay.getMonth( ) + '/'
25                         + currentDay.getDay( ) + '/'
26                         + currentDay.getYear( ) );
27
28     currentDay.nextDay( );
29     System.out.println( "The next day is "
30                         + currentDay.getMonth( ) + '/'
31                         + currentDay.getDay( ) + '/'
32                         + currentDay.getYear( ) );
33  }
34 }
```

EXAMPLE 3.2 Calling Methods

Figure 3.3
Output of Example 3.2

```
Independence day is in month 7
The current day for graduation is 15
The revised day for graduation is 12
The current day is 9/30/2008
The next day is 10/1/2008
```

For now, we'll postpone discussion of the last two methods in the class API, *toString* and *equals*, except to say that their functions, respectively, are to convert the object data to a printable format and to compare the object data to another object's data. All classes provide these methods.

Skill Practice
with these end-of-chapter questions

3.19.1 Multiple Choice Exercises

Questions 2, 3, 4, 5, 9, 10

3.19.8 Technical Writing

Questions 69, 70

3.4 Using Object References

As we have mentioned, an object reference points to the data of an object. The object reference and the object data are distinct entities. Any object can have more than one object reference pointing to it, or an object can have no object references pointing to it.

In Example 3.3, two *SimpleDate* object references, *hireDate* and *promotionDate*, are declared and their objects are instantiated at lines 9 and 14. Lines 10–12 and 15–18 output the respective data member values of *hireDate* and *promotionDate*. Then, line 20 uses the assignment operator to copy the object reference *hireDate* to the object reference *promotionDate*. After line 20, both object references have the same value and therefore point to the location of the same object, as shown in Figure 3.4. The second object, with values (9, 28, 2007), no longer has an object reference pointing to it and is now marked for **garbage collection**. The **garbage collector**, which is part of the Java Virtual Machine, releases the memory allocated to objects that

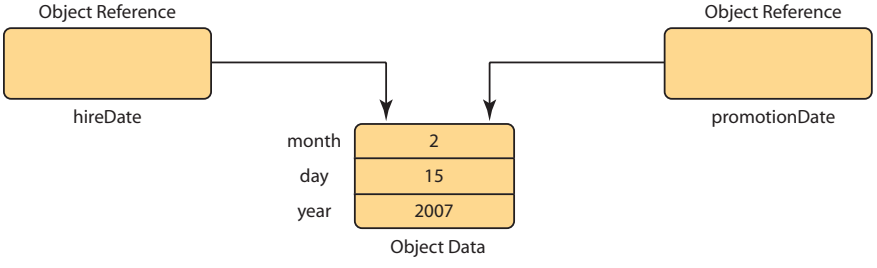


Figure 3.4
Two Object References Pointing to the Same Object

no longer have an object reference pointing to them. Lines 23–25 and 26–29 output the respective data member values of *hireDate* and *promotionDate* again. These are now identical, as shown in Figure 3.5.

```
1 /*  A demonstration of object reference assignment
2     Anderson, Franceschi
3 */
4
5 public class ObjectReferenceAssignment
6 {
7   public static void main( String [ ] args )
8   {
9      SimpleDate hireDate = new SimpleDate( 2, 15, 2007 );
10     System.out.println( "hireDate is " + hireDate.getMonth( )
11                         + "/" + hireDate.getDay( )
12                         + "/" + hireDate.getYear( ) );
13
14     SimpleDate promotionDate = new SimpleDate( 9, 28, 2007 );
15     System.out.println( "promotionDate is "
16                         + promotionDate.getMonth( )
17                         + "/" + promotionDate.getDay( )
18                         + "/" + promotionDate.getYear( ) );
19
20     promotionDate = hireDate;
21     System.out.println( "\nAfter assigning hireDate "
22                         + "to promotionDate:" );
23     System.out.println( "hireDate is " + hireDate.getMonth( )
24                         + "/" + hireDate.getDay( )
25                         + "/" + hireDate.getYear( ) );
26     System.out.println( "promotionDate is "
27                         + promotionDate.getMonth( )
28                         + "/" + promotionDate.getDay( )
29                         + "/" + promotionDate.getYear( ) );
30  }
31 }
```

EXAMPLE 3.3 Demonstrating Object Reference Assignments

When an object reference is first declared, but has not yet been assigned to an object, its value is a special literal value, **null**.

If you attempt to call a method using an object reference whose value is *null*, Java generates either a compiler error or a run-time error called an

```
hireDate is 2/15/2007
promotionDate is 9/28/2007

After assigning hireDate to promotionDate:
hireDate is 2/15/2007
promotionDate is 2/15/2007
```

Figure 3.5
Output of Example 3.3

exception. The exception is a *NullPointerException* and results in a series of messages printed on the Java console indicating where in the program the *null* object reference was used. Line 10 of Example 3.4 will generate a compiler error, as shown in Figure 3.6, because *aDate* has not been instantiated.

```
1 /*  A demonstration of trying to use a null object reference
2     Anderson, Franceschi
3 */
4
5 public class NullReference
6 {
7   public static void main( String [ ] args )
8   {
9      SimpleDate aDate;
10     aDate.setMonth( 5 );
11  }
12 }
```

EXAMPLE 3.4 Attempting to Use a *null* Object Reference

```
NullReference.java:10: variable aDate might not have been initialized
     aDate.setMonth( 5 );
     ^

1 error
```

Figure 3.6
Compiler error from Example 3.4

COMMON ERROR TRAP

Using a *null* object reference to call a method will generate either a compiler error or a *NullPointerException* at run time. Be sure to instantiate an object before attempting to use the object reference.

Java does not provide support for explicitly deleting an object. One way to indicate to the garbage collector that your program is finished with an object is to set its object reference to *null*. Obviously, once an object reference has the value *null*, it can no longer be used to call methods.

```
1 /*  A demonstration of trying to use a null object reference
2     Anderson, Franceschi
3 */
4
5 public class NullReference2
6 {
7   public static void main( String [ ] args )
8   {
9      SimpleDate independenceDay = new SimpleDate( 7, 4, 1776 );
10     System.out.println( "The month of independenceDay is "
11                         + independenceDay.getMonth( ) );
12
13     independenceDay = null;  // set object reference to null
14     // attempt to use object reference
15     System.out.println( "The month of independenceDay is "
16                         + independenceDay.getMonth( ) );
17  }
18 }
```

EXAMPLE 3.5 Another Attempt to Use a *null* Object Reference

Example 3.5 shows a *NullPointerException* being generated at run time. Line 9 instantiates the *independenceDay* object, and lines 10–11 print the month. Line 13 assigns *null* to the object reference and lines 15–16 attempt to print the month again. As Figure 3.7 shows, a *NullPointerException* is generated. Notice that the console message indicates the name of the application class (*NullReference2*), the method *main*, and the line number *15*, where the exception occurred. The JVM often prints additional lines in the message, depending on where in your program the error occurred.

Figure 3.7
Output of Example 3.5

```
The month of independenceDay is 7
Exception in thread "main" java.lang.NullPointerException
   at NullReference2.main(NullReference2.java:15)
```

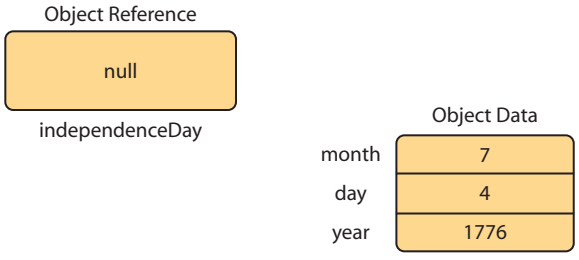


Figure 3.8
The *independenceDay* Object Reference Set to *null*

Figure 3.8 shows the *independenceDay* object reference and object data after setting the object reference to *null*.

3.5 Programming Activity 1: Calling Methods

Let's put this all together with a sample program that uses a *SimpleDate* object. In this Programming Activity, we'll use a program that displays the values of the object data as you instantiate the object and call the methods of the class.

In the Chapter 3 Programming Activity 1 folder on the CD-ROM accompanying this book, you will find three source files: *SimpleDate.java, SimpleDateClient.java,* and *Pause.java.* Copy all the *.java* and *.class* files to a directory on your computer. Note that all files should be in the same directory.

Open the *SimpleDateClient.java* source file. You'll notice that the class already contains some source code. Your job is to fill in the blanks. Search for five asterisks in a row (*****). This will position you to the places in the source code where you will add your code. This section of code is shown in Figure 3.9.

Notice that line 15 is a declaration of a *SimpleDate* object reference, *dateObj.* You will use this object reference for instantiating an object and for calling the methods of the *SimpleDate* class.

In the source file, you should see nine commented lines that instruct you to instantiate the object or call a method. You will also notice that there are eight lines that look like this:

```
// animate( "message" );
```

Figure 3.9
Partial Listing of *SimpleDateClient.java*

```
13   private int animationPause = 2; // 2 seconds between animations
14
15   SimpleDate dateObj; // declare Date object reference
16
17   public void workWithDates( )
18   {
19     animate( "dateObj reference declared" );
20
21     /***** Add your code here *****/
22     /**** 1. Instantiate dateObj using an empty argument list  */
23
24
25     //animate( "Instantiated dateObj - empty argument list" );
26
27     /***** 2. Set the month to the month you were born */
28
29
30     //animate( "Set month to birth month" );
31
32
33     /***** 3. Set the day to the day of the month you were born */
34
35
36     //animate( "Set day to birth day" );
37
38
39     /***** 4. Set the year to the year you were born */
40
41
42     //animate( "Set year to birth year" );
43
44
45     /***** 5. Call the nextDay method */
46
47
48     //animate( "Set the date to the next day" );
49
50
51     /***** 6. Set the day to 32, an illegal value */
52
53
54     //animate( "Set day to 32" );
55
56
57     /***** 7. Set the month to 13, an illegal value */
58
59
60     //animate( "Set month to 13" );
61
62
63     /***** 8. Assign the value null to dateObj */
64
65
66     //animate( "Set object reference to null" );
67
68
69     /***** 9. Attempt to set the month to 1 */
70
71   }
```

These lines are calls to an *animate* method in this class that displays the object reference and the object data after you have executed your code. The *message* is a *String* literal that describes what action your code just took. The *animate* method will display the message, as well as the object data. Note that when you call a method in the same class, you don't use an object reference and dot notation.

To complete the Programming Activity, write the requested code on the line between the numbered instruction and the *animate* method call. Then **uncomment** (remove the two slashes from) the *animate* method call.

For example, after you've written the code for the first instruction, lines 22 through 25 should look like this. The line you write is shown in bold.

```
/* 1. Instantiate a dateObj using empty argument list */

dateObj = new SimpleDate( );
animate( "Instantiated dateObj - empty argument list" );
```

Compile and run the code and you will see a window that looks like the one in Figure 3.10.

As you can see, the *dateObj* reference points to the *SimpleDate* object, and the *month*, *day*, and *year* instance variables have been assigned default values.

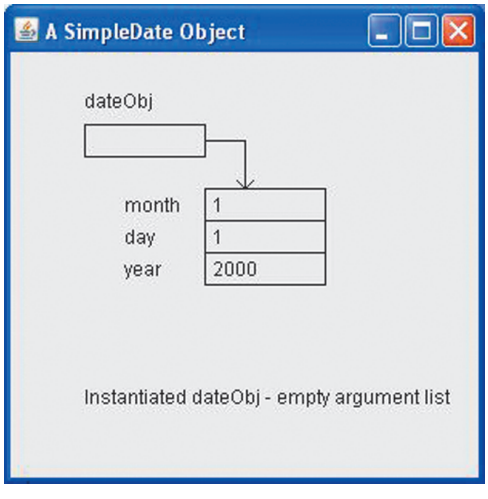


Figure 3.10
Programming Activity 1 Output

Write the code for the remaining instructions, compiling and running the program after completing each task. The program will display the changes you make to the object data.

The pause between animations is set by default to two seconds. To change the pause time, change the value assigned to *animationPause* on line 13 to the number of seconds you would like to pause between animations.

DISCUSSION QUESTIONS ?

1. After instructions 6 and 7 have executed, why do the day and month values get set to 1?
2. At the end of the execution of the program, a *NullPointerException* is generated. Which statement in the program causes this error? Explain why.

3.6 The Java Class Library

Java provides more than 2,000 predefined classes that you can use to add functionality to your program. In this chapter, we'll discuss a few commonly used Java classes:

- *String*, which provides a data type for character strings, along with methods for searching and manipulating strings
- *Random*, which generates random numbers
- *Scanner*, which provides methods for reading input from the Java console
- *System* and *PrintStream*, which provide data members and methods for printing data on the Java console
- *DecimalFormat* and *NumberFormat*, which allow you to format numbers for output
- *Math*, which provides methods for performing mathematical operations
- Object wrappers, which provide an object equivalent to primitive data types so they can be used in your program as if they were objects
- *JOptionPane*, which allows you to use dialog boxes to display messages to the user or to get input from the user

The Java classes are arranged in **packages**, grouped according to functionality.

TABLE 3.2 Commonly Used Java Packages

Package	Categories of Classes
java.lang	Basic functionality common to many programs, such as the *String* class, *Math* class, and object wrappers for the primitive data types
java.awt	Graphics classes for drawing and using colors, and old-style user interface components
javax.swing	New-style user interface components that have a consistent look and feel across platforms
java.text	Classes for formatting numeric output
java.util	the *Scanner* class, the *Random* class, and other miscellaneous classes
java.io	Classes for reading from and writing to files

Table 3.2 describes some of the Java packages that we will cover in this book. You can find more details on these classes on Sun Microsystems' Java website *http://java.sun.com*.

Many of the commonly used classes, such as *String* and *Math*, reside in the *java.lang* package. Any class in the *java.lang* package is automatically available to your program.

To use a class that is not in the *java.lang* package, you need to tell the compiler in which package the class resides; in other words, you need to tell the compiler where to find the class definition. To do this, you include an **import** statement in your program. The *import* statement is inserted at the top of the program after your introductory comments, but before the *class* statement that begins the program. (Yes, we've defined classes already in the programs we wrote in the previous two chapters.)

For example, if you want to use the *DecimalFormat* class to format a floating-point number for output, you would import the *DecimalFormat* class from the *java.text* package as follows:

```
import java.text.DecimalFormat;
```

If you're using more than one class from a package, you can import the whole package by using an asterisk in place of the class name, as follows:

```
import java.text.*;
```

3.7 The *String* Class

As we've discussed, Java provides the *char* primitive data type, which stores one character. Almost every program, however, needs a data type that stores more than one character. Programs need to process names, addresses, or labels of many kinds. For example, many programs involve a login procedure where the user has to enter a user ID and a password. The program reads the user ID and password, compares them to values stored in a database, and allows the user to continue only if the user ID and password match the database values.

To handle this type of data, Java provides a *String* class. Because the *String* class is part of the *java.lang* package, it is automatically available to any Java program and you do not need to use the *import* statement. The *String* class provides several constructors, as well as a number of methods to manipulate, search, compare, and concatenate *String* objects.

Let's look at two of the *String* class constructors shown in Table 3.3. Example 3.6 shows how to use these two constructors in a program.

```
1 /* Demonstrating the String methods
2    Anderson, Franceschi
3 */
4 public class StringDemo
5 {
6   public static void main ( String [ ] args )
7   {
8     String s1 = new String( "OOP in Java " );
9     System.out.println( "s1 is: " + s1 );
10    String s2 = "is not that difficult. ";
11    System.out.println( "s2 is: " + s2 );
12
13    String s3 = s1 + s2; // new String is s1, followed by s2
14    System.out.println( "s1 + s2 returns: " + s3 );
15
16    System.out.println( "s1 is still: " + s1 ); // s1 is unchanged
```

TABLE 3.3 *String* Class Constructors

***String* Class Constructor Summary**
`String( String str )` allocates a *String* object with the value of *str*, which can be a *String* object or a *String* literal.
`String( )` allocates an empty *String* object.

```
17      System.out.println( "s2 is still: " + s2 ); // s2 is unchanged
18
19      String greeting1 = "Hi"; // instantiate greeting1
20      System.out.println( "\nThe length of " + greeting1 + " is "
21                          + greeting1.length( ) );
22
23      String greeting2 = new String( "Hello" ); // instantiate greeting2
24      int len = greeting2.length( );  // len will be assigned 5
25      System.out.println( "The length of " + greeting2 + " is " + len );
26
27      String empty = new String( );
28      System.out.println( "The length of the empty String is "
29                          + empty.length( ) );
30
31      String greeting2Upper = greeting2.toUpperCase( );
32      System.out.println( );
33      System.out.println( greeting2 + " converted to upper case is "
34                          + greeting2Upper );
35
36      String invertedName = "Lincoln, Abraham";
37
38      int comma = invertedName.indexOf( ',' ); // find the comma
39      System.out.println( "\nThe index of " + ',' + " in "
40                          + invertedName + " is " + comma );
41
42      // extract all characters up to comma
43      String lastName = invertedName.substring( 0, comma );
44      System.out.println( "Dear Mr. " + lastName );
45    }
46  }
```

EXAMPLE 3.6 Demonstrating *String* Methods

When this program runs, it will produce the output shown in Figure 3.11.

```
s1 is: OOP in Java
s2 is: is not that difficult.
s1 + s2 returns: OOP in Java is not that difficult.
s1 is still: OOP in Java
s2 is still: is not that difficult.

The length of Hi is 2
The length of Hello is 5
The length of the empty String is 0

Hello converted to upper case is HELLO

The index of , in Lincoln, Abraham is 7
Dear Mr. Lincoln
```

Figure 3.11
Output from Example 3.6

The first constructor

```
String( String str )
```

allocates a *String* object and sets its value to the sequence of characters in the argument *str*, which can be a *String* object or a *String* literal. Line 8 instantiates the *String s1* and sets its value to "OOP in Java". Similarly, line 23 instantiates a *String* named *greeting2*, and assigns it the value "Hello".

The second constructor

```
String( )
```

creates an empty *String*, in other words, a *String* containing no characters. You can add characters to the *String* later. This constructor will come in handy in programs where we build up our output, piece by piece. Line 27 uses the second constructor to instantiate an empty *String* named *empty*.

Additionally, because *Strings* are used so frequently in programs, Java provides special support for instantiating *String* objects without explicitly using the *new* operator. We can simply assign a *String* literal to a *String* object reference. Lines 10 and 19 assign *String* literals to the *s2* and *greeting1* *String* references.

Java also provides special support for appending a *String* to the end of another *String* through the **concatenation operator** (+) and the **shortcut version of the concatenation operator** (+=). This concept is illustrated in Example 3.6. Lines 8–11 declare, instantiate, and print two *String* objects, *s1* and *s2*. Line 13 concatenates *s1* and *s2* and the resulting *String* is assigned to the *s3 String* reference, which is printed at line 14. Finally, we output *s1* and *s2* again at lines 16 and 17 to illustrate that their values have not changed.

Note that the *String* concatenation operator is the same character as the addition arithmetic operator. In some cases, we need to make clear to the compiler which operator we want to use. For example, this statement uses both the *String* concatenation operator and the addition arithmetic operator:

```
System.out.println( "The sum of 1 and 2 is " + ( 1 + 2 ) );
```

Notice that we put *1 + 2* inside parentheses to let the compiler know that we want to add two *ints* using the addition arithmetic operator (+). The addition will be performed first because of the higher operator precedence of parentheses. Then it will become clear to the compiler that the other +

TABLE 3.4 *String* Methods

***String* Class Method Summary**	
Return value	**Method name and argument list**
`int`	`length( )`
	returns the length of the *String*
`String`	`toUpperCase( )`
	converts all letters in the *String* to uppercase
`String`	`toLowerCase( )`
	converts all letters in the *String* to lowercase
`char`	`charAt( int index )`
	returns the character at the position specified by *index*
`int`	`indexOf( String searchString )`
	returns the index of the beginning of the first occurrence of *searchString* or –1 if *searchString* is not found
`int`	`indexOf( char searchChar )`
	returns the index of the first occurrence of *searchChar* in the *String* or –1 if *searchChar* is not found
`String`	`substring( int startIndex, int endIndex )`
	returns a substring of the *String* object beginning at the character at index *startIndex* and ending at the character at index *endIndex – 1*

operator is intended to be a *String* concatenation operator because its operands are a *String* and an *int*.

Some useful methods of the *String* class are summarized in Table 3.4.

The length Method

The *length* method returns the number of characters in a *String*. Sometimes, the number of characters in a user ID is limited, for example, to eight, and this method is useful to ensure that the length of the ID does not exceed the limit.

The *length* method is called using a *String* object reference and the dot operator, as illustrated in lines 21, 24, and 29 of Example 3.6. At lines 21 and 29, the *length* method is called inside an output statement and the respective return values from the *length* method are output. At line 24, we call the *length* method for the *greeting2* object and assign the return value to the *int* variable *len*. Then at line 25, we output the value of the variable *len*. As shown in Figure 3.11, the length of "*Hi*" is 2, the length of "*Hello*" is 5, and the length of the empty *String* is 0.

The toUpperCase and toLowerCase Methods

The *toUpperCase* method converts all the letters in a *String* to uppercase, while the *toLowerCase* method converts all the letters in a *String* to lowercase. Digits and special characters are unchanged.

At line 31 of Example 3.6, the *toUpperCase* method is called using the object reference *greeting2*, and the return value is assigned to a *String* named *greeting2Upper*, which is then printed at lines 33 and 34.

The indexOf Methods

The *indexOf* methods are useful for searching a *String* to see if specific *Strings* or characters are in the *String*. The methods return the location of the first occurrence of a single *char* or the first character of a *String*.

The location, or **index**, of any character in a *String* is counted from the first position in the *String*, which has the index value of 0. Thus in this *String*,

```
String greeting = "Ciao";
```

the *C* is at index 0; the *i* is at index 1; the *a* is at index 2; and the *o* is at index 3. Because indexes begin at 0, the maximum index in a *String* is 1 less than the number of characters in the *String*. So the maximum index for *greeting* is `greeting.length( )-1`, which is 3.

In Example 3.6, line 38 retrieves the index of the first comma in the *String* *invertedName* and assigns it to the *int* variable *comma*; the value of *comma*, here 7, is then output at lines 39 and 40.

The charAt and substring Methods

The *charAt* and *substring* methods are useful for extracting either a single *char* or a group of characters from a *String*.

The *charAt* method returns the character at a particular index in a *String*. One of the uses of this method is for extracting just the first character of a *String*, which might be advantageous when prompting the user for an answer to a question. For example, we might ask users if they want to play again. They can answer "y," "yes," or "you bet!"

Our only concern is whether the first character is a *y*, so we could use this method to put the first character of their answer into a *char* variable. Assuming the user's answer was previously assigned to a *String* variable named *answerString*, we would use the following statement to extract the first character of *answerString*:

```
char answerChar = answerString.charAt( 0 );
```

In Chapter 5, we'll see how to test whether *answerChar* is a *y*.

The *substring* method returns a group of characters, or **substring**, from a *String*. The original *String* is unchanged. As arguments to the *substring* method, you specify the index at which to start extracting the characters and the index of the first character not to extract. Thus, the *endIndex* argument is one position past the last character to extract. We know this sounds a little awkward, but setting up the arguments this way actually makes the method easier to use, as we will demonstrate.

In Example 3.6, we want to extract the last name in the *String invertedName*. Line 38 finds the index of the comma and assigns it to the *int* variable *comma*, then line 43 extracts the substring from the first character (index 0) to the index of the comma (which conveniently won't extract the comma), and assigns it to the *String* variable *lastName*. When the variable *lastName* is output at line 44, its value is *Lincoln*, as shown in Figure 3.11.

When you are calculating indexes and the number of characters to extract, be careful not to specify an index that is not in the *String*, because that will generate a run-time error, *StringIndexOutOfBoundsException*.

COMMON ERROR TRAP

Specifying a negative start index or a start index past the last character of the *String* will generate a *StringIndexOutOfBoundsException*. Specifying a negative end index or an end index greater than the length of the *String* will also generate a *StringIndexOutOfBoundsException*.

REFERENCE POINT

You can read more about the *String* class on Sun Microsystems' Java website *http://java.sun.com*.

3.8 Formatting Output with the *DecimalFormat* Class

In a computer program, numbers represent a real-life entity, for instance, a price or a winning percentage. Floating-point numbers, however, are calculated to many decimal places and, as a result of some computations, can end up with more significant digits than our programs need. For example, the price of an item after a discount could look like 3.466666666666666,

TABLE 3.5 A *DecimalFormat* Constructor and the *format* Method

***DecimalFormat* Class Constructor**	
`DecimalFormat( String pattern )`	instantiates a *DecimalFormat* object with the output *pattern* specified in the argument.
The *format* Method	
Return value	**Method name and argument list**
`String`	`format( double number )` returns a *String* representation of *number* formatted according to the *DecimalFormat* object used to call the method.

when all we really want to display is $3.47; that is, with a leading dollar sign and two significant digits after the decimal point. The *DecimalFormat* class allows you to specify the number of digits to display after the decimal point and to add dollar signs, commas, and percentage signs (%) to your output.

The *DecimalFormat* class is part of the *java.text* package, so to use the *DecimalFormat* class, you should include the following *import* statement in your program:

```
import java.text.DecimalFormat;
```

We can instantiate a *DecimalFormat* object using a simple constructor that takes a *String* object as an argument. This *String* object represents how we want our formatted number to look when it's printed. The API for that constructor is shown in Table 3.5.

The pattern that we use to instantiate the *DecimalFormat* object consists of special characters and symbols and creates a "picture" of how we want the number to look when printed. Some of the more commonly used symbols and their meanings are listed in Table 3.6.

```
1 /* Demonstrating the DecimalFormat class
2      Anderson, Franceschi
3 */
4
5 // import the DecimalFormat class from the java.text package;
6 import java.text.DecimalFormat;
7
```

TABLE 3.6 Special Characters for *DecimalFormat* Patterns

Common Pattern Symbols for a *DecimalFormat* Object	
Symbol	**Meaning**
0	Required digit. Do not suppress 0's in this position.
#	Optional digit. Do not print a leading or terminating digit that is 0.
.	Decimal point.
,	Comma separator.
$	Dollar sign.
%	Multiply by 100 and display a percentage sign.

```
 8 public class DemoDecimalFormat
 9 {
10   public static void main( String [ ] args )
11   {
12      // first, instantiate a DecimalFormat object specifying a
13      // pattern for currency
14      DecimalFormat pricePattern = new DecimalFormat( "$#0.00" );
15
16      double price1 = 78.66666666;
17      double price2 = 34.5;
18      double price3 = .3333333;
19      int price4 = 3;
20      double price5 = 100.23;
21
22      // then print the values using the pattern
23      System.out.println( "The first price is: "
24                  + pricePattern.format( price1 ) );
25      System.out.println( "\nThe second price is: "
26                  + pricePattern.format( price2 ) );
27      System.out.println( "\nThe third price is: "
28                  + pricePattern.format( price3 ) );
29      System.out.println( "\nThe fourth price is: "
30                  + pricePattern.format( price4 ) );
31      System.out.println( "\nThe fifth price is: "
32                  + pricePattern.format( price5 ) );
33
34      // instantiate another new DecimalFormat object
```

```
35      // for printing percentages
36      DecimalFormat percentPattern = new DecimalFormat( "#0.0%" );
37
38      double average = .980;
39      System.out.println( "\nThe average is: "
40                  + percentPattern.format( average ) );
41      // notice that the average is multiplied by 100
42      // to print a percentage.
43
44
45      // now instantiate another new DecimalFormat object
46      // for printing time as two digits
47      DecimalFormat timePattern = new DecimalFormat( "00" );
48
49      int hours = 5, minutes = 12, seconds = 0;
50      System.out.println( "\nThe time is "
51                  + timePattern.format( hours ) + ":"
52                  + timePattern.format( minutes ) + ":"
53                  + timePattern.format( seconds ) );
54
55      // now instantiate another DecimalFormat object
56      // for printing numbers in the millions.
57      DecimalFormat bigNumber = new DecimalFormat( "#,###" );
58
59      int millions = 1234567;
60      System.out.println( "\nmillions is "
61                  + bigNumber.format( millions ) );
62  }
63 }
```

EXAMPLE 3.7 Demonstrating the *DecimalFormat* Class

Once we have instantiated a *DecimalFormat* object, we format a number by passing it as an argument to the *format* method, shown in Table 3.5. Example 3.7 demonstrates the use of the *DecimalFormat* patterns and calling the *format* method. The output for this program is shown in Figure 3.12.

In Example 3.7, line 14 instantiates the *DecimalFormat* object, *pricePattern*, which will be used to print prices. In the pattern

```
"$#0.00"
```

the first character of this pattern is the dollar sign ($), which we want to precede the price. The # character specifies that leading zeroes should not be printed. The 0 specifies that there should be at least one digit to the left of the decimal point. If there is no value to the left of the decimal point,

```
The first price is: $78.67

The second price is: $34.50

The third price is: $0.33

The fourth price is: $3.00

The fifth price is: $100.23

The average is: 98.0%

The time is 05:12:00

millions is 1,234,567
```

Figure 3.12
Output from Example 3.7

then print a zero. The two 0's that follow the decimal point specify that two digits should be printed to the right of the decimal point; that is, if more than two digits are to the right of the decimal point, round to two digits; if the last digit is a 0, print the zero, and if there is no fractional part to the number, print two zeroes. Using this pattern, we see that in lines 23–24, *price1* is rounded to two decimal places. In lines 25–26, *price2* is printed with a zero in the second decimal place.

In lines 29–30, we print *price4*, which is an integer. The *format* method API calls for a *double* as the argument; however, because all numeric data types can be promoted to a *double*, any numeric data type can be sent as an argument. The result is that two zeroes are added to the right of the decimal point.

Finally, we use the *pricePattern* pattern to print *price5* in lines 31–32, which needs no rounding or padding of extra digits.

Next, line 36 instantiates a *DecimalFormat* object, *percentPattern*, for printing percentages to one decimal point (`"#0.0%"`). Lines 38–40 define the variable *average*, then print it using the *format* method. Notice that the *format* method automatically multiplies the value of *average* by 100.

REFERENCE POINT

You can read more about the *DecimalFormat* class on Sun Microsystems' Java website *http://java.sun.com*.

Line 47 defines another pattern, `"00"`, which is useful for printing the time with colons between the hour, minutes, and seconds. When the time is printed on lines 50–53, the hours, minutes, and seconds are padded with a leading zero, if necessary.

Line 57 defines our last pattern, `"#,###"`, which can be used to insert commas into integer values in the thousands and above. Lines 60–61 print the variable *millions* with commas separating the millions and thousands digits. Notice that the pattern is extrapolated for a number that has more digits than the pattern.

3.9 Generating Random Numbers with the *Random* Class

Random numbers come in handy for many operations in a program, such as rolling dice, dealing cards, timing the appearance of a nemesis in a game, or other simulations of seemingly random events.

There's one problem in using random numbers in programs, however: Computers are **deterministic**. In essence, this means that given a specific input to a specific set of instructions, a computer will always produce the same output. The challenge, then, is generating random numbers while using a deterministic system. Many talented computer scientists have worked on this problem, and some innovative and complex solutions have been proposed.

The *Random* class, which is in the *java.util* package, uses a mathematical formula to generate a sequence of numbers, feeding the formula a **seed** value, which determines where in that sequence the set of random numbers will begin. As such, the *Random* class generates numbers that appear to be, but are not truly, random. These numbers are called **pseudorandom** numbers, and they work just fine for our purposes.

Table 3.7 shows a constructor for the *Random* class and a method for retrieving a random integer. The default constructor creates a random number generator using a seed value. Once the random number generator is created, we can ask for a random number by calling the *nextInt* method. Other methods, *nextDouble*, *nextBoolean*, *nextByte*, and *nextLong*, which are not shown in Table 3.7, return a random *double*, *boolean*, *byte*, or *long* value, respectively.

To demonstrate how to use the random number generator, let's take rolling a die as an example. To simulate the roll of a six-sided die, we need to

TABLE 3.7 A *Random* Class Constructor and the *nextInt* Method

***Random* Class Constructor**	
`Random( )`	Creates a random number generator.
The *nextInt* Method	
Return value	**Method name and argument list**
`int`	`nextInt( int number )` returns a random integer ranging from 0 up to, but not including, *number* in uniform distribution

simulate random occurrences of the numbers 1 through 6. If we call the *nextInt* method with an argument of 6, it will return an integer between 0 and 5. To get randomly distributed numbers from 1 to 6, we can simply add 1 to the value returned by the *nextInt* method. Thus, if we have instantiated a *Random* object named *random*, we can generate random numbers from 1 to 6, by calling the *nextInt* method in this way:

```
int die = random.nextInt( 6 ) + 1;
```

In general, then, if we want to generate random numbers from n to m, we should call the *nextInt* method with the number of random values we need $(m - n + 1)$, and then add the first value of our sequence (n) to the returned value. Thus, this statement generates a random number between 10 and 100 inclusive:

```
int randomNumber =  random.nextInt( 100 - 10 + 1 ) + 10;
```

Line 18 of Example 3.8 will generate a random number between 20 and 200 inclusive.

```
1 /*  A demonstration of the Random class
2     Anderson, Franceschi
3 */
4 import java.util.Random;
5
6 public class RandomNumbers
7 {
8   public static void main( String [ ] args )
```

```
 9   {
10     Random random = new Random( );
11
12     // simulate the roll of a die
13     int die = random.nextInt( 6 ) + 1;
14     System.out.println( "\nThe die roll is " + die );
15
16     // generate a random number between 20 and 200
17     int start = 20, end = 200;
18     int number = random.nextInt( end - start + 1 ) + start;
19     System.out.println( "\nThe random number between " + start
20                         + " and " +  end  + " is " + number );
21   }
22 }
```

EXAMPLE 3.8 A Demonstration of the *Random* Class

Figure 3.13
Output from Example 3.8

```
The die roll is 2

The random number between 20 and 200 is 117
```

REFERENCE POINT

You can read more about the *Random* class on Sun Microsystems' Java website *http://java.sun.com*.

When the *RandomNumbers* program executes, it will produce output similar to the window shown in Figure 3.13. The output will vary from one execution of the program to the next because different random numbers will be generated.

3.10 Input from the Console Using the *Scanner* Class

As our programs become more complex, we will need to allow the users of our programs to input data. User input can be read into your program in several ways:

- from the Java console
- from a dialog box
- from a file
- through a Graphical User Interface (GUI)

The Java class library provides classes for all types of data input. In this chapter, we will concentrate on two ways to input data: from the Java console and from a dialog box. In Chapter 6 and Chapter 11, we explore how to

input data from a file, and in Chapter 12, we learn how to input data through a GUI.

The *Scanner* class provides methods for reading *byte*, *short*, *int*, *long*, *float*, *double*, and *String* data types from the Java console. These methods are shown in Table 3.8.

TABLE 3.8 Selected Methods of the *Scanner* Class

A *Scanner* Class Constructor

`Scanner( InputStream dataSource )`

creates a *Scanner* object that will read from the *InputStream dataSource*. To read from the keyboard, we will use the predefined *InputStream System.in*.

Selected Methods of the *Scanner* Class

Return value	Method name and argument list
`byte`	`nextByte( )` returns the next input as a *byte*
`short`	`nextShort( )` returns the next input as a *short*
`int`	`nextInt( )` returns the next input as an *int*
`long`	`nextLong( )` returns the next input as a *long*
`float`	`nextFloat( )` returns the next input as a *float*
`double`	`nextDouble( )` returns the next input as a *double*
`boolean`	`nextBoolean( )` returns the next input as a *boolean*
`String`	`next( )` returns the next token in the input line as a *String*
`String`	`nextLine( )` returns the input line as a *String*

The *Scanner* class is defined in the *java.util* package, so your programs will need to include the following *import* statement:

```
import java.util.Scanner;
```

In order to use the *Scanner* class, you must first instantiate a *Scanner* object and associate it with a data source. We will use the *System.in* input stream, which by default is tied to the keyboard. Thus, our data source for input will be *System.in*. The following statement will instantiate a *Scanner* object named *scan* and associate *System.in* as the data source.

```
Scanner scan = new Scanner( System.in );
```

Once the *Scanner* object has been instantiated, you can use it to call any of the *next. . .* methods to input data from the Java console. The specific *next. . .* method you call depends on the type of input you want from the user. Each of the *next. . .* methods returns a value from the input stream. You will need to assign the return value from the *next. . .* methods to a variable to complete the data input. Obviously, the data type of the variable must match the data type of the value returned by the *next. . .* method.

The *next. . .* methods just perform input. They do not tell the user what data to enter. Before calling any of the *next* methods, therefore, you need to prompt the user for the input you want. You can print a prompt using *System.out.print*, which is similar to using *System.out.println*, except that the cursor remains after the printed text, rather than advancing to the next line.

When writing a prompt for user input, keep several things in mind. First, be specific. If you want the user to enter his or her full name, then your prompt should say just that:

```
Please enter your first and last names.
```

If the input should fall within a range of values, then tell the user which values will be valid:

```
Please enter an integer between 0 and 10.
```

Also keep in mind that users are typically not programmers. It's important to phrase a prompt using language the user understands. Many times, programmers write a prompt from their point of view, as in this bad prompt:

```
Please enter a String:
```

Users don't know, and don't care, about *Strings* or any other data types, for that matter. Users want to know only what they need to enter to get the program to do its job.

When your prompts are clear and specific, the user makes fewer errors and therefore feels more comfortable using your program.

Line 13 of Example 3.9 prompts the user to enter his or her first name. Line 14 captures the user input and assigns the word entered by the user to the *String* variable *firstName*, which is printed in line 15. Similarly, line 17 prompts for the user's age; line 18 captures the integer entered by the user and assigns it to the *int* variable *age*, and line 19 outputs the value of *age*. Reading other primitive data types follows the same pattern. Line 21 prompts for the user's grade point average (a *float* value). Line 22 captures the number entered by the user and assigns it to the *float* variable *gpa*, and line 23 outputs the value of *gpa*.

SOFTWARE ENGINEERING TIP

Provide the user with clear prompts for input. Prompts should be phrased using words the user understands and should describe the data requested and any restrictions on valid input values.

```
1 /*  A demonstration of reading from the console using Scanner
2     Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class DataInput
8 {
9   public static void main( String [ ] args )
10  {
11     Scanner scan = new Scanner( System.in );
12
13     System.out.print( "Enter your first name > " );
14     String firstName = scan.next( );
15     System.out.println( "Your name is " + firstName );
16
17     System.out.print( "\nEnter your age as an integer > " );
18     int age = scan.nextInt( );
19     System.out.println( "Your age is " + age );
20
21     System.out.print( "\nEnter your GPA > " );
22     float gpa = scan.nextFloat( );
23     System.out.println( "Your GPA is " + gpa );
24  }
25 }
```

EXAMPLE 3.9 Reading from the Console using *Scanner*

Figure 3.14
Data Input with Example 3.9

```
Enter your first name > Syed
Your name is Syed

Enter your age as an integer > 21
Your age is 21

Enter your GPA > 3.875
Your GPA is 3.875
```

When this program executes, the prompt is printed on the console and the cursor remains at the end of the prompt. Figure 3.14 shows the output when these statements are executed and the user enters *Syed*, presses *Enter*, enters 21, presses *Enter*, and enters 3.875, and presses *Enter* again.

SOFTWARE ENGINEERING TIP

End your prompts with some indication that input is expected, and include a trailing space for better readability.

The methods *nextByte*, *nextShort*, *nextLong*, *nextDouble*, and *nextBoolean* can be used with the same pattern as *next*, *nextInt*, and *nextFloat*.

Note that we end our prompt with a space, an angle bracket, and another space. The angle bracket indicates that we are waiting for input, and the spaces separate the prompt from the input. Without the trailing space, the user's input would immediately follow the prompt, which is more difficult to read, as you can see in Figure 3.15.

As you review Table 3.8, you may notice that the *Scanner* class does not provide a method for reading a single character. To do this, we can use the *next* method, which returns a *String*, then extract the first character from the *String* using the *charAt(0)* method call, as shown in Example 3.10. Line 14 inputs a *String* from the user and assigns it to the *String* variable *initialS*, then line 15 assigns the first character of *initialS* to the *char* variable *initial*; *initial* is then output at line 16 as shown in Figure 3.16.

Figure 3.15
Prompt and Input Running Together

```
Enter your age as an integer >21
```

```
1 /*  A demonstration of how to get character input using Scanner
2     Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class CharacterInput
8 {
9   public static void main( String [ ] args )
10  {
11     Scanner scan = new Scanner( System.in );
12
13     System.out.print( "Enter your middle initial > " );
14     String initialS = scan.next( );
15     char initial = initialS.charAt( 0 );
16     System.out.println( "Your middle initial is " + initial );
17  }
18 }
```

EXAMPLE 3.10 Using *Scanner* for Character Input

```
Enter your middle initial > A
Your middle initial is A
```

Figure 3.16
Output of Example 3.10

A *Scanner* object divides its input into sequences of characters called **tokens**, using **delimiters**. The default delimiters are the standard **white-space** characters, which among others include the space, tab, and newline characters. The complete set of Java whitespace characters is shown in Table 3.9.

By default, when a *Scanner* object tokenizes the input, it skips leading whitespace, then builds a token composed of all subsequent characters until it encounters another delimiter. Thus, if you have this code,

```
System.out.print( "Enter your age as an integer > " );
int age = scan.nextInt( );
```

and the user types, for example, three spaces and a tab, *21*, and a newline:

```
<space><space><space><tab>21<newline>
```

TABLE 3.9 Java Whitespace Characters

Character	Unicode equivalents
space	`\u00A0`, `\u2007`, `\u202F`
tab	`\u0009`, `\u000B`
line feed	`\u000A`
form feed	`\u000C`
carriage return	`\u000D`
file, group, unit, and record separators	`\u001C`, `\u001D`, `\u001E`, `\u001F`

then the *Scanner* object skips the three spaces and the tab, starts building a token with the character *2*, then adds the character *1* to the token, and stops building the token when it encounters the *newline*. Thus, *21* is the resulting token, which the *nextInt* method returns into the *age* variable.

An input line can contain more than one token. For example, if we prompt the user for his or her name and age, and the user enters the following line, then presses *Enter:*

```
<tab>Jon<space>Olsen,<space>21<space>
```

then, the leading whitespace is skipped and the *Scanner* object creates three tokens:

- *Jon*
- *Olsen,*
- *21*

Note that commas are not whitespace, so the comma is actually part of the second token. To input these three tokens, your program would use two calls to the *next* method to retrieve the two *String* tokens and a call to *nextInt* to retrieve the age.

To capture a complete line of input from the user, we use the method *nextLine*. Example 3.11 shows how *nextLine* can be used in a program. Figure 3.17 shows a sample run of the program with the user entering data.

```
1 /*  A demonstration of using Scanner's nextLine method
2     Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class InputALine
8 {
9   public static void main( String [ ] args )
10  {
11    Scanner scan = new Scanner( System.in );
12
13    System.out.print( "Enter a sentence > " );
14    String sentence  = scan.nextLine( );
15    System.out.println( "You said: \"" + sentence + "\"" );
16  }
17 }
```

EXAMPLE 3.11 Using the *nextLine* Method

```
Enter a sentence > Scanner is useful.
You said: "Scanner is useful."
```

Figure 3.17
Output of Example 3.11

If the user's input (that is, the next token) does not match the data type of the *next. . .* method call, then an *InputMismatchException* is generated and the program stops. Figure 3.18 demonstrates Example 3.9 when the program calls the *nextInt* method and the user enters a letter, rather than an

REFERENCE POINT

You can read more about the *Scanner* class on Sun Microsystems' Java website: *http://java.sun.com*.

```
Enter your first name > Sarah
Your name is Sarah

Enter your age as an integer > a
Exception in thread "main" java.util.InputMismatchException
   at java.util.Scanner.throwFor(Unknown Source)
   at java.util.Scanner.next(Unknown Source)
   at java.util.Scanner.nextInt(Unknown Source)
   at java.util.Scanner.nextInt(Unknown Source)
   at DataInput.main(DataInput.java:18)
```

Figure 3.18
An Exception When Input Is Not the Expected Data Type

integer. In Chapter 6, we show you how to avoid this exception, and in Chapter 11, we show you how to intercept the exception and recover from it.

If the user doesn't type anything when prompted, or if the user types some characters but doesn't press *Enter,* the program will simply wait until the user does press *Enter.*

Skill Practice
with these end-of-chapter questions

3.19.1 Multiple Choice Exercises

Questions 1, 11

3.19.2 Reading and Understanding Code

Questions 14, 15, 16

3.19.3 Fill In the Code

Questions 24, 25, 26, 27

3.19.4 Identifying Errors in Code

Questions 36, 37, 38, 39, 43

3.19.5 Debugging Area

Questions 45, 49

3.19.6 Write a Short Program

Questions 50, 51, 52

3.11 Calling *Static* Methods and Using *Static* Class Variables

Classes can also define **static methods,** which can be called without instantiating an object. These are also called **class methods**. The API of these methods has the keyword *static* before the return type:

```
static dataType methodName( arg1, arg2, . . . )
```

One reason a class may define *static* methods is to provide some quick, one-time functionality without requiring the client to instantiate an object. For example, dialog boxes typically pop up only once in a program. Creating an

object for a dialog box, when it is used only once, is a waste of memory and processor time. We'll see later in this chapter how it's possible to create dialog boxes and to perform mathematical calculations without creating an object.

Class, or *static*, methods are invoked using the class name, rather than an object reference, as in the following syntax:

```
ClassName.staticMethodName( argumentList );
```

For example, in this statement:

```
absValue = Math.abs( someNumber );
```

the class name is *Math*, and the *static* method is *abs*, which returns the absolute value of the argument (*someNumber*). We use the class name rather than an object reference, because *static* methods can be called without instantiating an object. Later in this chapter, we will explore some *static* methods of the *Math* class in greater detail.

Because *static* methods can be called without an object being instantiated, *static* methods cannot access the instance variables of the class (because instance variables are object data and exist only after an object has been instantiated). *Static* methods can access **static data**, however, and classes often declare *static* data to be used with *static* methods. *Static* data belong to the class, rather than to a particular object, or instance, of the class.

A common use of *static* class variables is to define constants for commonly used values or for parameters for the *static* class methods. For example, as we'll discuss in Chapter 4, the *Color* class provides *static* constants that can be assigned to a *Color* object reference.

Like *static* methods, *static* constants are also accessed using the class name and dot operator, as in this syntax:

```
ClassName.staticConstant
```

Thus, the *static* constant representing the color blue can be accessed this way:

```
Color.BLUE
```

At first, this may appear to go against our earlier discussion of encapsulation and the restrictions on clients directly accessing object data. Remember we said that the client needed to use accessor ("gets") and mutator ("sets") methods to access object data. The reasoning behind encapsulation is to protect the object data from corruption by the client. However, in this

case, the *static* data is constant, so the client is unable to change it. For the client, directly accessing the class constant is easier and faster than calling a method.

3.12 Using *System.in* and *System.out*

In order to print program output to the screen, we have been using statements like

```
System.out.println( "The value of b is " + b );
```

and

```
System.out.print( "Enter your first name > " );
```

And to instantiate a *Scanner* object, we used this statement:

```
Scanner scan = new Scanner( System.in );
```

It is now time to look at these statements in depth and understand them completely.

System is an existing Java class in the *java.lang* package. One of its fields is a *static* constant, *out*, which represents the Java console by default. Another of its fields is a *static* constant, *in*, which represents the keyboard by default. Because *in* and *out* are *static*, we refer to them using the class name, *System*, and the dot notation:

```
System.out
System.in
```

Table 3.10 shows these static constants as well as the *static exit* method, which can be used to terminate a program. Calling *System.exit( )* at the end of a program is optional. After the last instruction is executed, the program will end in any case. However, the *exit* method of the *System* class can be useful if you want to stop execution at a place other than the end of the program.

System.out is an object of the *PrintStream* class, which is also an existing Java class; it can be found in the *java.io* package. The *out* object refers to the **standard output device**, which by default is the Java console.

The methods *print* and *println* belong to the *PrintStream* class and take arguments of any primitive type, a *String*, or an object reference. The only

TABLE 3.10 *Static* Constants of the *System* Class and the *exit* Method

Constant	Value
`in`	*static* constant that represents the standard input stream, by default the keyboard
`out`	*static* constant that represents the standard output stream, by default the Java console
A Useful *System* Method	
Return value	**Method name and argument list**
`void`	`exit( int exitStatus )` *static* method that terminates the Java Virtual Machine. A value of 0 for *exitStatus* indicates a normal termination. Any other values indicate abnormal termination and are used to signal that the program ended because an error occurred.

difference between *print* and *println* is that *println* will also print a *newline* character after it writes the output. Table 3.11 shows some methods of the *PrintStream* class, which can be used with *System.out*.

TABLE 3.11 *PrintStream* Methods for Use with *System.out*

Useful *PrintStream* Methods	
Return value	**Method name and argument list**
`void`	`print( argument )` prints *argument* to the standard output device. The *argument* can be any primitive data type, a *String* object, or another object reference.
`void`	`println( argument )` prints *argument* to the standard output device, then prints a *newline* character. The *argument* can be any primitive data type, a *String*, or another object reference.
`void`	`println( )` prints a *newline* character. This method is useful for skipping a line in the program's output.

Example 3.12 demonstrates various ways to use the *print* and *println* methods:

```
1 /*  Testing the print and println methods
2     Anderson, Franceschi
3 */
4
5 public class PrintDemo
6 {
7   public static void main( String [ ] args )
8   {
9     System.out.println( "Combine the arguments using concatenation" );
10    System.out.println( "A double: " + 23.7 + ", and an int: " + 78 );
11
12    System.out.print( "\nJava is case sensitive: " );
13    System.out.println( 'a' + " is different from " + 'A' );
14
15    System.out.println( "\nCreate a variable and print its value" );
16    String s = new String( "The grade is" );
17    double grade = 3.81;
18    System.out.println( s + " " + grade  );
19
20    System.out.println( );  // skip a line
21    SimpleDate d = new SimpleDate( 4, 5, 2009 );
22    System.out.println( "Explicitly calling toString, d is "
23                        + d.toString( ) );
24    System.out.println( "Implicitly calling toString, d is " + d );
25
26    System.exit( 0 );  // optional
27  }
28 }
```

EXAMPLE 3.12 Demonstrating the *print* and *println* Methods

Lines 10 and 13 show how *print* or *println* can be used with various data types such as *double*, *int*, and *char*. Variables and expressions can also be used instead of literals, as shown in line 18, where the *String s* and the *double* variable *grade* are output.

We can also print objects. All classes have a *toString* method, which converts the object data to a *String* for printing. The *toString* method is called automatically whenever an object is used as a *String*. Notice that our *SimpleDate* class, introduced earlier in the chapter, had a *toString* method that returned the object data as a *String* in the format *mm/dd/yyyy*.

```
Combine the arguments using concatenation
A double: 23.7, and an int: 78

Java is case sensitive: a is different from A

Create a variable and print its value
The grade is 3.81

Explicitly calling toString, d is 4/5/2009
Implicitly calling toString, d is 4/5/2009
```

Figure 3.19
The Output from Example 3.12

The *toString* method's API is

```
String toString( )
```

After the *SimpleDate* object reference *d* is instantiated at line 21, it is printed at lines 22–23 and again at line 24. At lines 22–23, the method *toString* is called explicitly; at line 24, it is called automatically. The output of Example 3.12 is shown in Figure 3.19. Finally, we terminate the program by calling the *exit* method of the *System* class.

REFERENCE POINT

You can read more about the *System* and *PrintStream* classes on Sun Microsystems' Java website *http://java.sun.com*.

3.13 The *Math* Class

The *Math* class is also part of the *java.lang* package. As such, it is automatically available to any Java program; you do not need to use the *import* statement. The *Math* class provides two *static* constants (*E* and *PI*), as well as a number of *static* methods that save the programmer from writing some complex mathematical code.

The two constants, *E* and *PI*, are both *doubles* and represent, respectively, ***e*** (the base of the natural logarithm, i.e., log e = 1) and ***pi***, the ratio of the circumference of a circle to its diameter. Approximate values of *e* and *pi*, as we know them, are 2.78 and 3.14, respectively. These constants are shown in Table 3.12.

Because *E* and *PI* are *static* data members of the *Math* class, they are referenced using the name of the *Math* class and the dot notation as follows:

```
Math.E
Math.PI
```

TABLE 3.12 *Static* Constants of the *Math* Class

Constant	Value
`E`	*e*, the base of the natural logarithm
`PI`	*pi*, the ratio of the circumference of a circle to its diameter

Useful methods of the *Math* class are shown in Table 3.13. All the methods of the *Math* class are *static*; so they are called using the class name, *Math*, and the dot notation as follows:

```
Math.abs( -5 )
```

TABLE 3.13 Useful Methods of the *Math* Class

Math Class Method Summary	
Return value	**Method name and argument list**
`dataTypeOfArg`	`abs( arg )` *static* method that returns the absolute value of the argument *arg*, which can be a *double*, *float*, *int*, or *long*.
`double`	`log( double a )` *static* method that returns the natural logarithm (in base e) of its argument, *a*. For example, log(1) returns 0 and log(*Math.E*) returns 1.
`dataTypeOfArgs`	`min( argA, argB )` *static* method that returns the smaller of the two arguments. The arguments can be *doubles*, *floats*, *ints*, or *longs*.
`dataTypeOfArgs`	`max( argA, argB )` *static* method that returns the larger of the two arguments. The arguments can be *doubles*, *floats*, *ints*, or *longs*.
`double`	`pow( double base, double exp )` *static* method that returns the value of *base* raised to the *exp* power.
`long`	`round( double a )` *static* method that returns the closest integer to its argument, *a*.
`double`	`sqrt( double a )` *static* method that returns the positive square root of *a*.

Example 3.13 demonstrates how the *Math* constants and the *abs* method can be used in a Java program. In lines 9 and 10, we print the values of *e* and *pi* using the *static* constants of the *Math* class. Then in lines 12 and 15, we call the *abs* method, which returns the absolute value of its argument. We then print the results in lines 13 and 16. The output of Example 3.13 is shown in Figure 3.20.

```
1 /*  A demonstration of the Math class methods and constants
2     Anderson, Franceschi
3 */
4
5 public class MathConstants
6 {
7   public static void main( String [ ] args )
8   {
9     System.out.println( "The value of e is " + Math.E );
10    System.out.println( "The value of pi is " + Math.PI );
11
12    double d1 = Math.abs( 6.7 ); // d1 will be assigned 6.7
13    System.out.println( "\nThe absolute value of 6.7 is " + d1 );
14
15    double d2 = Math.abs( -6.7 ); // d2 will be assigned 6.7
16    System.out.println( "\nThe absolute value of -6.7 is " + d2 );
17  }
18 }
```

EXAMPLE 3.13 *Math* Class Constants and the *abs* Method

```
The value of e is 2.718281828459045
The value of pi is 3.141592653589793

The absolute value of 6.7 is 6.7

The absolute value of -6.7 is 6.7
```

Figure 3.20
Output from Example 3.13

The operation and usefulness of most *Math* class methods are obvious. But several methods—*pow, round*, and *min/max*—require a little explanation.

The pow Method

Example 3.14 demonstrates how some of these *Math* methods can be used in a Java program.

```
1 /*  A demonstration of some Math class methods
2     Anderson, Franceschi
3 */
4
5 public class MathMethods
6 {
7   public static void main( String [ ] args )
8   {
9     double d2 = Math.log( 5 );
10    System.out.println( "\nThe log of 5 is " + d2 );
11
12    double d4 = Math.sqrt( 9 );
13    System.out.println( "\nThe square root of 9 is " + d4 );
14
15    double fourCubed = Math.pow( 4, 3 );
16    System.out.println( "\n4 to the power 3 is " + fourCubed );
17
18    double bigNumber = Math.pow( 43.5, 3.4 );
19    System.out.println( "\n43.5 to the power 3.4 is " + bigNumber );
20  }
21 }
```

EXAMPLE 3.14 A Demonstration of Some *Math* Class Methods

The *Math* class provides the *pow* method for raising a number to a power. The *pow* method takes two arguments, the first is the base and the second is the exponent.

Although the argument list for the *pow* method specifies that the base and the exponent are both *doubles,* you can, in fact, send arguments of any numeric type to the *pow* method because all numeric types can be promoted to a *double*. No matter what type the arguments are, however, the return value is always a *double*. Thus, when line 15 calls the *pow* method with two integer arguments, the value of *fourCubed* will be *64.0*. If you prefer that the return value be 64, you can cast the return value to an *int*.

Line 18 shows how to use the *pow* method with arguments of type *double*. The output of Example 3.14 is shown in Figure 3.21.

```
The log of 5 is 1.6094379124341003

The square root of 9 is 3.0

4 to the power 3 is 64.0

43.5 to the power 3.4 is 372274.65827529586
```

Figure 3.21
Output from Example 3.14

The round Method

The *round* method converts a *double* to its nearest integer using these rules:

- any factional part .0 to .4 is rounded down
- any fractional part .5 and above is rounded up

Lines 9–13 in Example 3.15 use the *round* method with various numbers. Figure 3.22 shows the output.

```
1 /*  A demonstration of the Math round method
2     Anderson, Franceschi
3 */
4
5 public class MathRounding
6 {
7   public static void main( String [ ] args )
8   {
9     System.out.println( "23.4 rounded is " + Math.round( 23.4 ) );
10    System.out.println( "23.49 rounded is " + Math.round( 23.49 ) );
11    System.out.println( "23.5 rounded is " + Math.round( 23.5 ) );
12    System.out.println( "23.51 rounded is " + Math.round( 23.51 ) );
13    System.out.println( "23.6 rounded is " + Math.round( 23.6 ) );
14  }
15 }
```

EXAMPLE 3.15 A Demonstration of the *Math round* method

```
23.4 rounded is 23
23.49 rounded is 23
23.5 rounded is 24
23.51 rounded is 24
23.6 rounded is 24
```

Figure 3.22
Output from Example 3.15

The min and max Methods

The *min* and *max* methods return the smaller or larger of their two arguments, respectively. Example 3.16 demonstrates how the *min* and *max* methods can be used in a Java program. Figure 3.23 shows the output. Thus the statement on line 9 of Example 3.16

```
int smaller = Math.min( 8, 2 );
```

will assign 2 to the *int* variable *smaller*. At line 12, a similar statement using the *max* method will assign 8 to the *int* variable *larger*.

```
1 /*  A demonstration of min and max Math class methods
2     Anderson, Franceschi
3 */
4
5 public class MathMinMaxMethods
6 {
7   public static void main( String [ ] args )
8   {
9     int smaller = Math.min( 8, 2 );
10    System.out.println( "The smaller of 8 and 2 is " + smaller );
11
12    int larger = Math.max( 8, 2 );
13    System.out.println( "The larger of 8 and 2 is " + larger );
14
15    int a = 8, b = 5, c = 12;
16    int tempSmaller = Math.min( a, b );   // find smaller of a & b
17    int smallest = Math.min( tempSmaller, c ); // compare result to c
18    System.out.println( "The smallest of " + a + ", " + b + ", and "
19                        + c + " is " + smallest );
20  }
21 }
```

EXAMPLE 3.16 A Demonstration of the *min* and *max* Methods

Figure 3.23
Output from Example 3.16

```
The smaller of 8 and 2 is 2
The larger of 8 and 2 is 8
The smallest of 8, 5, and 12 is 5
```

The *min* method can also be used to compute the smallest of three variables. After declaring and initializing the three variables (*a*, *b*, and *c*) at line 15, we assign to a temporary variable named *tempSmaller* the smaller of the first two variables, *a* and *b*, at line 16. Then, at line 17, we compute the smaller of *tempSmaller* and the third variable, *c*, and assign that value to the *int* variable *smallest*, which is output at lines 18 and 19.

The pattern for finding the largest of three numbers is similar, and we leave that as an exercise at the end of the chapter.

REFERENCE POINT

You can read more about the *Math* class on Sun Microsystems' Java website *http://java.sun.com*.

Skill Practice
with these end-of-chapter questions

3.19.1 Multiple Choice Exercises

Questions 6, 7, 8, 13

3.19.2 Reading and Understanding Code

Questions 17, 18, 19, 20, 21, 22, 23

3.19.3 Fill In the Code

Questions 28, 29, 30, 31, 32, 34

3.19.4 Identifying Errors in Code

Questions 40, 41, 42

3.19.5 Debugging Area

Questions 46, 47, 48

3.19.6 Write a Short Program

Questions 53, 54

CODE IN ACTION

To see a step-by step illustration of how to instantiate an object and call both instance and *static* methods, look for the Chapter 3 Flash movie on the CD accompanying this book. Click on the link for Chapter 3 to view the movie.

3.14 Formatting Output with the *NumberFormat* Class

Like the *DecimalFormat* class, the *NumberFormat* class can also be used to format numbers for output. The *NumberFormat* class, however, provides specialized *static* methods for creating objects specifically for formatting currency and percentages.

The *NumberFormat* class is part of the *java.text* package, so you need to include the following *import* statement at the top of your program.

```
import java.text.NumberFormat;
```

The static methods of the *NumberFormat* class to format currency and percentages are shown in Table 3.14.

As you can see from the first two method headers, their return type is a *NumberFormat* object. These *static* methods, called **factory methods**, are used instead of constructors to create objects. Thus, instead of using the *new* keyword and a constructor, we will call one of these methods to create our *formatting* object.

The *getCurrencyInstance* method returns a formatting object that reflects the local currency. In the United States, that format is a leading dollar sign and two digits to the right of the decimal place. The *getPercentInstance* method returns a formatting object that prints a fraction as a percentage by multiplying the fraction by 100, rounding to the nearest whole percent, and adding a percent sign (%).

TABLE 3.14 Useful Methods of the *NumberFormat* Class

***NumberFormat* Method Summary**	
Return value	**Method name and argument list**
`NumberFormat`	`getCurrencyInstance( )` *static* method that creates a format object for money.
`NumberFormat`	`getPercentInstance( )` *static* method that creates a format object for percentages.
`String`	`format( double number )` returns a *String* representation of *number* formatted according to the object used to call the method.

We then use the *format* method from the *NumberFormat* class to display a value either as money or a percentage. The *format* method takes one argument, which is the variable or value that we want to print; it returns the formatted version of the value as a *String* object, which we can then print.

Example 3.17 is a complete program illustrating how to use these three methods.

```
/*  Demonstration of currency and percentage formatting
    using the NumberFormat class.
    Anderson, Franceschi
*/

// we need to import the NumberFormat class from java.text
import java.text.NumberFormat;

public class DemoNumberFormat
{
  public static void main( String [ ] args )
  {
     double winningPercentage = .675;
     double price = 78.9;

     // get a NumberFormat object for printing a percentage
     NumberFormat percentFormat = NumberFormat.getPercentInstance( );

     // call format method using the NumberFormat object
     System.out.print( "The winning percentage is " );
     System.out.println( percentFormat.format( winningPercentage ) );

     // get a NumberFormat object for printing currency
     NumberFormat priceFormat = NumberFormat.getCurrencyInstance( );

     // call format method using the NumberFormat object
     System.out.println( "\nThe price is: "
                         + priceFormat.format( price ) );
  }
}
```

EXAMPLE 3.17 Demonstrating the *NumberFormat* Class

The output of this program is shown in Figure 3.24.

Figure 3.24
Output from Example 3.17

```
The winning percentage is 68%

The price is: $78.90
```

3.15 The *Integer*, *Double*, and Other Wrapper Classes

In Chapter 2, we discussed primitive data types and how they can be used in a program. In this chapter, we've discussed classes and class methods and how useful and convenient classes are in representing and encapsulating data into objects.

Most programs use a combination of primitive data types and objects. Some class methods, however, will accept only objects as arguments, so we need some way to convert a primitive data type into an object. Conversely, there are times when we need to convert an object into a primitive data type. For example, let's say we have a Graphical User Interface where we ask users to type their age into a text box or a dialog box. We expect the age to be an *int* value; however, text boxes and dialog boxes return their values as *Strings*. To perform any calculations on an age in our program, we will need to convert the value of that *String* object into an *int*.

For these situations, Java provides **wrapper classes**. A wrapper class "wraps" the value of a primitive type, such as *double* or *int*, into an object. These wrapper classes define an instance variable of that primitive data type, and also provide useful constants and methods for converting between the objects and the primitive data types. Table 3.15 lists the wrapper classes for each primitive data type.

All these classes are part of the *java.lang* package. So, the *import* statement is not needed in order to use them in a program.

To convert a primitive *int* variable to an *Integer* wrapper object, we can instantiate the *Integer* object using the *Integer* constructor.

```
int intPrimitive = 42;
Integer integerObject = new Integer( intPrimitive );
```

However, because this is a common operation, Java provides special support for converting between a primitive numeric type and its wrapper class. Instead of using the *Integer* constructor, we can simply assign the *int* vari-

TABLE 3.15 Wrapper Classes for Primitive Data Types

Primitive Data Type	Wrapper Class
double	Double
float	Float
long	Long
int	Integer
short	Short
byte	Byte
char	Character
boolean	Boolean

able to an *Integer* object reference. Java will automatically provide the conversion for us. This conversion is called **autoboxing**. In Example 3.18, the conversion is illustrated in lines 9 and 10. The *int* variable, *intPrimitive*, and the *Integer* object, *integerObject*, are output at lines 12 and 13 and have the same value (42). The output is shown in Figure 3.25.

Similarly, when an *Integer* object is used as an *int*, Java also provides this conversion, which is called **unboxing**. Thus, when we use an *Integer* object in an arithmetic expression, the *int* value is automatically used. Line 15 of Example 3.18 uses the *Integer* object *integerObject* in an arithmetic expression, adding the *Integer* object to the *int* variable *intPrimitive*. As shown in Figure 3.25, the result is the same as if both operands were *int* variables.

Similar operations are possible using other numeric primitives and their associated wrapper classes.

In addition to automatic conversions between primitive types and wrapper objects, the *Integer* and *Double* classes provide methods, shown in Table 3.16, that allow us to convert between primitive types and objects of the *String* class.

The *parseInt*, *parseDouble*, and *valueOf* methods are *static* and are called using the *Integer* or *Double* class name and the dot notation. The *parse* methods convert a *String* to a primitive type, and the *valueOf* methods convert a *String* to a wrapper object. For example, line 18 of Example 3.18 converts the *String* "*76*" to the *int* value 76. Line 19 converts the *String* "*76*" to an *Integer* object.

```
1 /*  A demonstration of the Wrapper classes and methods
2     Anderson, Franceschi
3 */
4
5 public class DemoWrapper
6 {
7   public static void main( String [ ] args )
8   {
9     int intPrimitive = 42;
10    Integer integerObject = intPrimitive;
11
12    System.out.println( "The int is " + intPrimitive );
13    System.out.println( "The Integer object is " + integerObject );
14
15    int sum = intPrimitive + integerObject;
16    System.out.println( "The sum is " + sum );
17
18    int i1 = Integer.parseInt( "76" );    // convert "76" to an int
19    Integer i2 = Integer.valueOf( "76" ); // convert "76" to Integer
20    System.out.println( "\nThe value of i1 is " + i1 );
21    System.out.println( "The value of i2 is " + i2 );
22
23    double d1 = Double.parseDouble( "58.32" );
24    Double d2 = Double.valueOf( "58.32" );
25    System.out.println( "\nThe value of d1 is " + d1 );
26    System.out.println( "The value of d2 is " + d2 );
27  }
28 }
```

EXAMPLE 3.18 A Demonstration of the Wrapper Classes

Figure 3.25
Output from Example 3.18

```
The int is 42
The Integer object is 42
The sum is 84

The value of i1 is 76
The value of i2 is 76

The value of d1 is 58.32
The value of d2 is 58.32
```

TABLE 3.16 Methods of the *Integer* and *Double* Wrapper Classes

Useful Methods of the *Integer* Wrapper Class

Return value	Method name and argument list
`int`	`parseInt( String s )` *static* method that converts the *String s* to an *int* and returns that value
`Integer`	`valueOf( String s )` *static* method that converts the *String s* to an *Integer* object and returns that object

Useful Methods of the *Double* Wrapper Class

Return value	Method name and argument list
`double`	`parseDouble( String s )` *static* method that converts the *String s* to a *double* and returns that value
`Double`	`valueOf( String s )` *static* method that converts the *String s* to a *Double* object and returns that object

Similarly, line 23 converts the *String* "*58.32*" to a *double*, and line 24 converts the same *String* to a *Double* object.

The usefulness of these wrappers will become clear in the next section of this chapter, where we discuss dialog boxes.

REFERENCE POINT

You can read more about the wrapper classes on Sun Microsystems' Java website *http://java.sun.com*.

3.16 Input and Output Using *JOptionPane* Dialog Boxes

Java provides the *JOptionPane* class for creating dialog boxes—those familiar pop-up windows that prompt the user to enter a value or notify the user of an error. The *JOptionPane* class is in the *javax.swing* package, so you will need to provide an *import* statement in any program that uses a dialog box.

TABLE 3.17 Input and Output Methods of the *JOptionPane* Class

Useful Methods of the *JOptionPane* Class	
Return value	**Method name and argument list**
`String`	`showInputDialog( Component parent, Object prompt )` *static* method that pops up an input dialog box, where *prompt* asks the user for input. Returns the characters typed by the user as a *String*.
`void`	`showMessageDialog( Component parent, Object message )` *static* method that pops up an output dialog box with *message* displayed

Most classes in the *javax.swing* package are designed for GUIs, but *JOptionPane* dialog boxes can be used in both GUI and non-GUI programs.

Table 3.17 lists some useful *JOptionPane static* methods.

The *showInputDialog* method is used for input, that is, for prompting the user for a value and inputting that value into the program. The *showMessageDialog* method is used for output, that is, for printing a message to the user. Although Java provides several constructors for dialog boxes, it is customary to create dialog boxes that will be used only once using the *static* methods and the *JOptionPane* class name.

Let's look first at the method *showInputDialog*, which gets input from the user. It takes two arguments: a parent component object and a prompt to display. At this point, our applications won't have a parent component object, so we'll always use *null* for that argument.

The second argument, the prompt, is usually a *String*, and lets the user know what kind of input our program needs. Next, notice that the return value of the *showInputDialog* method is a *String*.

Example 3.19 shows how the *showInputDialog* method is used to retrieve user input through a dialog box.

```
1 /* Using dialog boxes for input and output of Strings
2     Anderson, Franceschi
3 */
4
5 import javax.swing.JOptionPane;
6
7 public class DialogBoxDemo1
```

```
 8 {
 9   public static void main( String [ ] args )
10   {
11     String name = JOptionPane.showInputDialog( null,
12                   "Please enter your first and last names" );
13     JOptionPane.showMessageDialog( null, "Hello, " + name );
14   }
15 }
```

EXAMPLE 3.19 Using Dialog Boxes with *Strings*

When lines 11 and 12 are executed, the dialog box in Figure 3.26 appears. The user types his or her name into the white box, then presses either the *Enter* key or clicks the *OK* button. At that time, the *showInputDialog* method returns a *String* representing the characters typed by the user, and that *String* is assigned to the variable *name*.

To output a message to the user, use the *showMessageDialog* method. The *showMessageDialog* method is similar to the *showInputDialog* method in that it takes a parent component object (*null* for now) and a *String* to display. Thus, in Example 3.19, line 13 uses the variable *name* to echo back to the user a greeting.

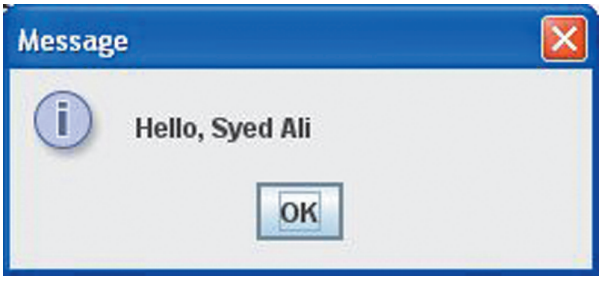
Notice that because the *showMessageDialog* is a method with a *void* return value, you call it as a standalone statement, rather than using the method call in an expression.

If the user typed "Syed Ali" when prompted for his name, the output dialog box shown in Figure 3.27 would appear.

To input an integer or any data type other than a *String*, however, you need to convert the returned *String* to the desired data type. Fortunately, as we saw in the previous section, you can do this using a wrapper class and its associated *parse* method, as Example 3.20 demonstrates.

Figure 3.26
Dialog Box Prompting for First and Last Names

Figure 3.27
Output Dialog Box from Example 3.19




```
1 /* Demonstrating dialog boxes for input and output of numbers
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JOptionPane;
6
7 public class DialogBoxDemo2
8 {
9   public static void main( String [ ] args )
10  {
11     String input = JOptionPane.showInputDialog( null,
12           "Please enter your age in years" );
13     int age = Integer.parseInt( input );
14     JOptionPane.showMessageDialog( null, "Your age is  " + age );
15
16     double average = Double.parseDouble(
17           JOptionPane.showInputDialog( null,
18           "Enter your grade point average between 0.0 and 4.0" ) );
19     JOptionPane.showMessageDialog( null, "Your average is "
20           + average );
21  }
22 }
```

EXAMPLE 3.20 Converting Input *Strings* to Numbers

Lines 11 and 12 pop up an input dialog box and assign the characters entered by the user to the *String input*. Line 13 uses the *parseInt* method of the *Integer* class to convert *input* to an integer, which is assigned to the *int* variable *age*. Line 14 then displays the value of *age* in an output dialog box.

Java programmers often combine multiple related operations into one statement in order to type less code and to avoid declaring additional variables. Lines 16, 17, and 18 illustrate this concept. At first it may look confusing, but if you look at the statement a piece at a time, it becomes clear what is happening. The *showInputDialog* method is called, returning a

String representing whatever the user typed into the dialog box. This *String* then becomes the argument passed to *parseDouble*, which converts the *String* to a *double*. Lines 19–20 display the value of *average* in another dialog box.

In this prompt, we included a range of valid values to help the user type valid input. However, including a range of values in your prompt does not prevent the user from entering other values. The *parseDouble* method will accept any *String* that can be converted to a numeric value. After your program receives the input, you will need to verify that the number entered is indeed within the requested range of values. In Chapter 6, we will show you techniques for verifying whether the user has entered valid values.

With either *Double.parseDouble* or *Integer.parseInt*, the value the user types must be convertible to the appropriate data type. If not, an exception is generated. For example, if the user enters *A* for the grade point average, the method generates a *NumberFormatException*. We'll discuss how you can intercept and handle exceptions in Chapter 11.

The various input and output dialog boxes from a sample run of Example 3.20 are shown in Figure 3.28.

REFERENCE POINT

You can read more about the *JOptionPane* class on Sun Microsystems' Java website *http://java.sun.com*.

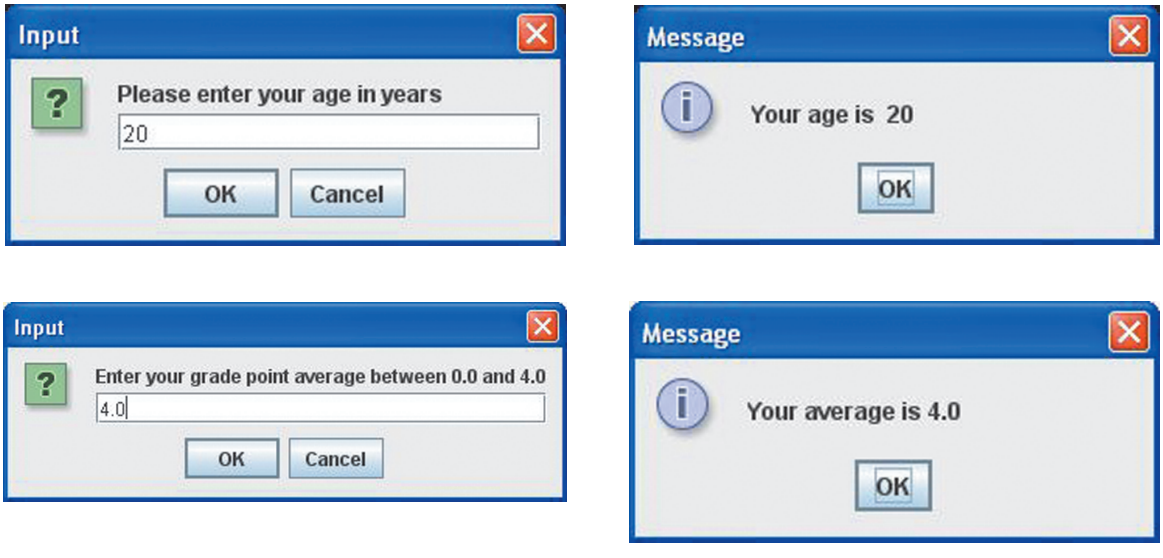


Figure 3.28
Dialog Boxes from Example 3.20

Skill Practice
with these end-of-chapter questions

3.19.1 Multiple Choice Exercises

Question 12

3.19.3 Fill In the Code

Question 33

3.19.3 Identifying Errors in Code

Question 35

3.19.5 Debugging Area

Question 44

3.19.6 Write a Short Program

Questions 55, 56

3.19.8 Technical Writing

Questions 71, 72

3.17 Programming Activity 2: Using Predefined Classes

In this Programming Activity, you will write a short program using some of the classes and methods discussed in this chapter. Plus, given the API of a method of an additional class, you will determine how to call the method. Your program will perform the following operations:

1. a. Prompt the user for his or her first name
   b. Print a message saying hello to the user
   c. Tell the user how many characters are in his or her name
2. a. Ask the user for the year of his or her birth
   b. Calculate and print the age the user will be this year
   c. Declare a constant for average life expectancy; set its value to 77.9
   d. Print a message that tells the user the percentage of his or her expected life lived so far

3. a. Generate a random number between 1 and 20

   b. Pop up a dialog box telling the user that the program is thinking of a number between 1 and 20 and ask for a guess

   c. Pop up a dialog box telling the user the number

To complete this Programming Activity, copy the contents of the Chapter 3 Programming Activity 2 folder on the CD-ROM accompanying this book. Open the *PracticeMethods.java* file and look for four sets of five asterisks (*****), where you will find instructions to write *import* statements and items 1, 2, and 3 for completing the Programming Activity.

Example 3.21 shows the *PracticeMethods.java* file, and Figures 3.29 and 3.30 show the output from a sample run after you have completed the

```
Enter your first name > Esmerelda
Hello Esmerelda
Your name has 9 letters

In what year were you born > 1990
This year, you will be 18
You have lived 23.1% of your life.
```

Figure 3.29
Console Output from a Sample Run of Programming Activity 2

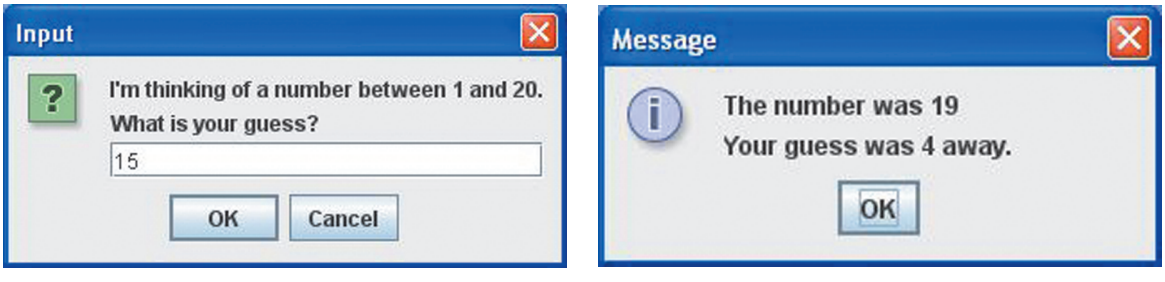


Figure 3.30
Dialog Boxes from a Sample Run of Programming Activity 2

Programming Activity. Because item 3 generates a random number, your output may be different.

```
/* Chapter 3 Programming Activity 2
   Calling class methods
   Anderson, Franceschi
*/

// ***** add your import statements here

public class PracticeMethods
{
  public static void main( String [ ] args )
  {
    //*****
    // 1.  a. Create a Scanner object to read from the console
    //     b. Prompt the user for his or her first name
    //     c. Print a message that says hello to the user
    //     d. Print a message that says how many letters
    //               are in the user's name
    // Your code goes here

    //*****
    // 2.  a. Skip a line, then prompt the user for the year
    //            of birth
    //     b. Calculate and print the age the user will be this year
    //     c. Declare a constant for average life expectancy,
    //            set its value to 77.9
    //     d. Print a message that tells the user the percentage
    //            of his or her expected life lived
    //       Use the DecimalFormat class to format the percentage

    //*****
    // 3.  a. Generate a random integer between 1 and 20
    //     b. Pop up an input dialog box and ask the user for a guess.
    //     c. Pop up an output dialog box telling the user the number
    //         and how far from the number the guess was (hint: use Math.abs)

  }
}
```

EXAMPLE 3.21 ***PracticeMethods.java***

? DISCUSSION QUESTIONS

1. Which methods of the *Scanner* class did you choose for reading the user's name and birth year? Explain your decisions.
2. How would you change your code to generate a random number between 10 and 20?

3.18 Chapter Summary

CHAPTER SUMMARY

- Object-oriented programming entails writing programs that use classes and objects. Using prewritten classes shortens development time and creates more reliable programs. Programs that use prewritten classes are called clients of the class.
- Benefits of object-oriented programming include encapsulation, reusability, and reliability.
- Classes consist of data, plus instructions that operate on that data. Objects of a class are created using the class as a template. Creating an object is called instantiating an object, and the object is an instance of the class. The *new* keyword is used to instantiate an object.
- The object reference is the variable name for an object and points to the data of the object.
- The data of a class are called instance variables or fields, and the instructions of the class are called methods. Methods of a class get or set the values of the data or provide other services of the class.
- The name of a method, along with its argument list and return value, is called the Application Programming Interface (API) of that method. Methods that are declared to be *public* can be called by any client of the class.
- By convention, class names in Java start with a capital letter. Method names, instance variables, and object names start with a lowercase letter. In all these names, embedded words begin with a capital letter.
- When your program makes a method call, control transfers to the instructions in the method until the method finishes executing. Then control is transferred back to your program.

CHAPTER SUMMARY

- Instance methods are called using the object reference and the dot notation.
- A constructor is called when an object is instantiated. A constructor has the same name as the class and its job is to initialize the object's data. Classes can have multiple constructors. Constructors have no return values.
- A method's data type is called the method's return type. If the data type is anything other than the keyword *void*, the method returns a value to the program. When a value-returning method finishes executing, its return value replaces the method call in the expression.
- Accessor methods, also called *gets*, allow clients to retrieve the current value of object data. Mutator methods, also called *sets*, allow clients to change the value of object data.
- When an object reference is first declared, its value is *null*. Attempting to use a *null* object reference to call a method generates an error.
- The garbage collector runs occasionally and deletes objects that have no object references pointing to them.
- Java packages are groups of classes arranged according to functionality. Classes in the *java.lang* packages are automatically available to Java programs. Other classes need to be imported.
- The *String* class can be used to create objects consisting of a sequence of characters. *String* constructors accept *String* literals, *String* objects, or no argument, which creates an empty *String*. The *length* method returns the number of characters in the *String* object. The *toUpperCase* and *toLowerCase* methods return a *String* in upper or lower case. The *charAt* method extracts a character from a *String*, while the *substring* method extracts a *String* from a *String*. The *indexOf* method searches a *String* for a character or substring.
- The *DecimalFormat* class, in the *java.text* package, formats numeric output. For example, you can specify the number of digits to display after the decimal point or add dollar signs and percentage signs (%).

- The *Random* class, in the *java.util* package, generates random numbers.
- The *Scanner* class, in the *java.util* package, provides methods for reading input from the Java console. Methods are provided for reading primitive data types and *Strings*.
- When prompting the user for input, phrase the prompt in language the user understands. Describe the data requested and any restrictions on valid input values
- *Static* methods, also called class methods, can be called without instantiating an object. *Static* methods can access only the *static* data of a class.
- *Static* methods are called using the class name and the dot notation.
- *System.out.println* prints primitive data types or a *String* to the Java console and adds a *newline* character. *System.out.println* with no argument skips a line. *System.out.print* prints the same data types to the Java console, but does not add a *newline*. Classes provide a *toString* method to convert objects to a *String* in order to be printed.
- The *Math* class provides *static* constants *PI* and *E* and *static* methods to perform common mathematical calculations, such as finding the maximum or minimum of two numbers, rounding values, and raising a number to a power.
- The *NumberFormat* class, in the *java.text* package, provides *static* methods for formatting numeric output as currency or a percentage.
- Wrapper classes provide an object interface for a primitive data type. The *Integer* and *Double* wrapper classes provide *static* methods for converting between *ints* and *doubles* and *Strings*.
- The *JOptionPane* class, in the *javax.swing* package, provides the *static* methods *showMessageDialog* for popping up an output dialog box and *showInputDialog* for popping up an input dialog box.

3.19 Exercises, Problems, and Projects

3.19.1 Multiple Choice Exercises

1. If you want to use an existing class from the Java class library in your program, what keyword should you use?
   - ❑ `use`
   - ❑ `import`
   - ❑ `export`
   - ❑ `include`
2. A constructor has the same name as the class name.
   - ❑ true
   - ❑ false
3. A given class can have more than one constructor.
   - ❑ true
   - ❑ false
4. What is the keyword used to instantiate an object in Java?
   - ❑ `make`
   - ❑ `construct`
   - ❑ `new`
   - ❑ `static`
5. In a given class named *Quiz*, there can be only one method with the name *Quiz*.
   - ❑ true
   - ❑ false
6. A *static* method is
   - ❑ a class method
   - ❑ an instance method
7. In the *Quiz* class, the *foo* method has the following API:

   ```
   public static double foo( float f )
   ```

   What can you say about *foo*?

- ❑ It is an instance method.
- ❑ It is a class field.
- ❑ It is a class method.
- ❑ It is an instance variable.

8. In the *Quiz* class, the *foo* method has the following API:

```
public static void foo( )
```

How would you call that method?

- ❑ `Quiz.foo( );`
- ❑ `Quiz.foo( 8 );`
- ❑ `Quiz( foo( ) );`

9. In the *Quiz* class, the *foo* method has the following API:

```
public double foo( int i, String s, char c )
```

How many arguments does *foo* take ?

- ❑ 0
- ❑ 1
- ❑ 2
- ❑ 3

10. In the *Quiz* class, the *foo* method has the following API:

```
public double foo( int i, String s, char c )
```

What is the return type of method *foo*?

- ❑ `double`
- ❑ `int`
- ❑ `char`
- ❑ `String`

11. *String* is a primitive data type in Java.

- ❑ true
- ❑ false

12. Which one of the following is not an existing wrapper class?

    - ❑ `Integer`
    - ❑ `Char`
    - ❑ `Float`
    - ❑ `Double`

13. What is the proper way of accessing the constant *E* of the *Math* class?

    - ❑ `Math.E( );`
    - ❑ `Math.E;`
    - ❑ `E;`
    - ❑ `Math( E );`

3.19.2 Reading and Understanding Code

14. What is the output of this code sequence?

```
String s = new String( "HI" );
System.out.println( s );
```

15. What is the output of this code sequence?

```
String s = "A" + "BC" + "DEF" + "GHIJ";
System.out.println( s );
```

16. What is the output of this code sequence?

```
String s = "Hello";
s = s.toLowerCase( );
System.out.println( s );
```

17. What is the output of this code sequence?

```
int a = Math.min( 5, 8 );
System.out.println( a );
```

18. What is the output of this code sequence?

```
System.out.println( Math.sqrt( 4.0 ) );
```

19. What is the output of this code sequence? (You will need to actually compile this code and run it in order to have the correct output.)

```
System.out.println( Math.PI );
```

20. What is the output of this code sequence?

```
double f = 5.7;
long i = Math.round( f );
System.out.println( i );
```

21. What is the output of this code sequence?

```
System.out.print( Math.round( 3.5 ) );
```

22. What is the output of this code sequence?

```
int i = Math.abs( -8 );
System.out.println( i );
```

23. What is the output of this code sequence?

```
double d = Math.pow( 2, 3 );
System.out.println( d );
```

3.19.3 Fill In the Code

24. This code concatenates the three *Strings* "Intro", "to", and "Programming" and outputs the resulting *String*. (Your output should be "Intro to Programming.")

```
String s1 = "Intro ";
String s2 = "to";
String s3 = " Programming";
// your code goes here
```

25. This code prints the number of characters in the *String* "Hello World."

```
String s = "Hello World";
// your code goes here
```

26. This code prompts the user for a *String*, then prints the *String* and the number of characters in it.

```
// your code goes here
```

27. This code uses only a single line *System.out.println* . . . statement in order to print

    "Welcome to Java Illuminated"

    on one line using (and only using) the following variables:

```
String s1 = "Welcome ";
String s2 = "to ";
String s3 = "Java ";
String s4 = "Illuminated";
// your code goes here
```

28. This code uses exactly four *System.out.print* statements in order to print

    "Welcome to Java Illuminated"

on the same output line.

```
// your code goes here
```

29. This code assigns the maximum of the values 3 and 5 to the *int* variable *i* and outputs the result.

```
int i;
// your code goes here
```

30. This code calculates the square root of 5 and outputs the result.

```
double d = 5.0;
// your code goes here
```

31. This code asks the user for two integer values, then calculates the minimum of the two values and prints it.

```
// your code goes here
```

32. This code asks the user for three integer values, then calculates the maximum of the three values and prints it.

```
// your code goes here
```

33. This code pops up a dialog box that prompts the user for an integer, converts the *String* to an *int*, adds 1 to the number, and pops up a dialog box that outputs the new value.

```
// your code goes here
```

34. This code asks the user for a *double*, then prints the square of this number.

```
// your code goes here
```

3.19.4 Identifying Errors in Code

35. Where is the error in this statement?

```
import text.NumberFormat;
```

36. Where is the error in this statement?

```
import java.util.DecimalFormat;
```

37. Where is the error in this code sequence?

```
String s = "Hello World";
system.out.println( s );
```

38. Where is the error in this code sequence?

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

39. Where is the error in this code sequence?

```
String s1 = "Hello";
String s2 = "ello";
String s = s1 - s2;
```

40. Where is the error in this code sequence?

```
short s = Math.round( 3.2 );
System.out.println( s );
```

41. Where is the error in this code sequence?

```
int a = Math.pow( 3, 4 );
System.out.println( a );
```

42. Where is the error in this code sequence?

```
double pi = Math( PI );
System.out.println( pi );
```

43. Where is the error in this code sequence?

```
String s = 'H';

System.out.println( "s is " + s );
```

3.19.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

44. You coded the following program in file *Test.java:*

```
public class Test
{
  public static void main( String [ ] args )
  {
    int a = 6;
    NumberFormat nf = NumberFormat.getCurrencyInstance( );
  }
}
```

When you compile, you get the following message:

```
Test.java: 6: cannot find symbol
  symbol : class NumberFormat
  location: class Test
  NumberFormat nf = NumberFormat.getCurrencyInstance( );
  ^
Test.java: 6: cannot find symbol
  symbol : variable NumberFormat
```

```
  location: class Test
  NumberFormat nf = NumberFormat.getCurrencyInstance( );
                                 ^
2 errors
```

Explain what the problem is and how to fix it.

45. You coded the following on lines 10–12 of class *Test.java:*

```
String s;                       // line 10
int l = s.length( );            // line 11
System.out.println( "length is " + l );     // line 12
```

When you compile, you get the following message:

```
Test.java:11: variable s might not have been initialized.
  int l = s.length( );   // line 11
          ^
1 error
```

Explain what the problem is and how to fix it.

46. You coded the following on lines 10 and 11 of class *Test.java:*

```
double d = math.sqrt( 6 );          // line 10
System.out.println( "d = " + d );   // line 11
```

When you compile, you get the following message:

```
Test.java: 10: cannot find symbol
  symbol : variable math
  location: class Test
double d = math.sqrt( 6 );  // line 10
               ^
  1 error
```

Explain what the problem is and how to fix it.

47. You coded the following on lines 10 and 11 of class *Test.java:*

```
double d = Math.PI( );              // line 10
System.out.println( "d = " + d );  // line 11
```

When you compile, you get the following message:

```
Test.java:10: cannot find symbol
symbol : method PI ( )
location: class java.lang.Math
  double d = Math.PI( );   // line 10
                  ^
1 error
```

Explain what the problem is and how to fix it.

48. You coded the following on lines 10 and 11 of class *Test.java:*

```
double d = Math.e;                  // line 10
System.out.println( "d = " + d );   // line 11
```

When you compile, you get the following message:

```
Test.java:10: cannot find symbol
  symbol : variable e
  location: class java.lang.Math
       double d = Math.e;                // line 10
                       ^
  1 error
```

Explain what the problem is and how to fix it.

49. You imported the *DecimalFormat* class and coded the following in the class *Test.java:*

```
double grade = .895;
DecimalFormat percent =
  new DecimalFormat( "#.0%" );

System.out.println( "Your grade is "
  + grade );
```

The code compiles properly and runs, but the result is not what you expected. You expect this output:

```
Your grade is 89.5%
```

But instead, the output is

```
Your grade is 0.895
```

Explain what the problem is and how to fix it.

3.19.6 Write a Short Program

50. Write a program that reads two words representing passwords from the Java console and outputs the number of characters in the smaller of the two. For example, if the two words are *open* and *sesame*, then the output should be *4*, the length of the shorter word, *open*.

51. Write a program that reads a name that represents a domain name from the Java console. Your program should then concatenate that name with *www.* and *.com* in order to form an Internet domain name and output the result. For instance, if the name entered by the user is *yahoo*, then the output will be *www.yahoo.com*.

52. Write a program that reads a word from the Java console. Your program should then output the same word, output the word in uppercase letters only, output that word in lowercase letters only, and then, at the end, output the original word.
53. Write a program that generates two random numbers between 0 and 100 and prints the smaller of the two numbers.
54. Write a program that takes a *double* as an input from the Java console, then computes and outputs the cube of that number.
55. Write a program that reads a filename from a dialog box. You should expect that the filename has one . (dot) character in it, separating the filename from the file extension. Retrieve the file extension and output it. For instance, if the user inputs *index.html*, you should output *html*; if the user inputs *MyClass.java*, you should output *java*.
56. Write a program that reads a full name (first name and last name) from a dialog box; you should expect the first name and the last name to be separated by a space. Retrieve the first name and output it.

3.19.7 Programming Projects

57. Write a program that reads three integer values from the Java console representing, respectively, a number of quarters, dimes, and nickels. Convert the total coin amount to dollars and output the result with a dollar notation.
58. Write a program that reads from the Java console the radius of a circle. Calculate and output the area and the perimeter of that circle. You can use the following formulas:

    area $= \pi * r^2$

    perimeter $= 2 * \pi * r$
59. Write a program that generates five random integers between 60 and 100 and calculates the smallest of the five numbers.
60. Write a program that generates three random integers between 0 and 50, calculates the average, and prints the result.
61. Write a program that reads two integers from the Java console: one representing the number of shots taken by a basketball player, the other representing the number of shots made by the same player.

Calculate the shooting percentage and output it with the percent notation.

62. Write a program that takes three *double* numbers from the Java console representing, respectively, the three coefficients *a*, *b*, and *c* of a quadratic equation. Solve the equation using the following formulas:

    $x1 = (-b + \text{square root } (b^2 - 4\ ac)) / (2a)$;

    $x2 = (-\text{b} - \text{square root } (b^2 - 4\ ac)) / (2a)$;

    Run your program on the following sample values:

    $a = 1.0, b = 3.0, c = 2.0$

    $a = 0.5, b = 0.5, c = 0.125$

    $a = 1.0, b = 3.0, c = 10.0$

    Discuss the results for each program run, in particular what happens in the last case.

63. Write a program that takes two numbers from the Java console representing, respectively, an investment and an interest rate (you will expect the user to enter a number such as .065 for the interest rate, representing a 6.5% interest rate). Your program should calculate and output (in $ notation) the future value of the investment in 5, 10, and 20 years using the following formula:

    future value = investment * (1 + interest rate)$^{\text{year}}$

    We will assume that the interest rate is an annual rate and is compounded annually.

64. Write a program that reads from the Java console the (*x*,*y*) coordinates for two points in the plane. You can assume that all numbers are integers. Using the *Point* class from Java (you may need to look it up on the Web), instantiate two *Point* objects with your input data, then output the data for both *Point* objects.

65. Write a program that reads a *char* from the Java console. Look up the *Character* class on the Web, in particular the method *getNumericValue*. Using the *getNumericValue* method, find the corresponding Unicode encoding number and output the character along with its corresponding Unicode value. Find all the Unicode values for characters a to z and A to Z.

66. Write a program that reads a telephone number from a dialog box; you should assume that the number is in this format: nnn-nnn-nnnn. You should output this same telephone number but with spaces instead of dashes, that is: nnn nnn nnnn.

67. Write a program that reads a sentence from a dialog box. The sentence has been encrypted as follows: only the first five even-numbered characters should be counted; all other characters should be discarded. Decrypt the sentence and output the result. For example, if the user inputs "Hiejlzl3ow", your output should be *Hello*.

68. Write a program that reads a commercial website URL from a dialog box; you should expect that the URL starts with *www* and ends with *.com*. Retrieve the name of the site and output it. For instance, if the user inputs *www.yahoo.com*, you should output *yahoo*.

3.19.8 Technical Writing

69. At this point, we have written and debugged many examples of code. When you compile a Java program with the Java compiler, you get a list of all the errors in your code. Do you like the Java compiler? Do the error messages it displays when your code does not compile help you determine what's wrong?

70. Computers, computer languages, and application programs existed before object-oriented programming. However, OOP has become an industry standard. Discuss the advantages of using OOP compared to using only basic data types in a program.

71. Explain and discuss a situation where you would use the method *parseInt* of the class *Integer*.

72. In addition to the basic data types (*int*, *float*, *char*, *boolean*, . . .), Java provides many prewritten classes, such as *Math*, *NumberFormat*, and *DecimalFormat*. Why is this an advantage? How does this impact the way a programmer approaches a programming problem in general?

3.19.9 Group Project (for a group of 1, 2, or 3 students)

73. Write a program that calculates a monthly mortgage payment; we will assume that the interest rate is compounded monthly.

You will need to do the following:

- ❑ Prompt the user for a *double* representing the annual interest rate.
- ❑ Prompt the user for the number of years the mortgage will be held (typical input here is 10, 15, or 30).
- ❑ Prompt the user for a number representing the mortgage amount borrowed from the bank.
- ❑ Calculate the monthly payment using the following formulas:
  - Monthly payment = (mIR * M) / (1 − (1 / (1 + mIR)$^{(12*nOY)}$)), where:
    - mIR = monthly interest rate = annual interest rate / 12
    - nOY = number of years
    - M = mortgage amount
- ❑ Output a summary of the mortgage problem, as follows:
  - the annual interest rate in percent notation
  - the mortgage amount in dollars
  - the monthly payment in dollars, with only two significant digits after the decimal point
  - the total payment over the years, with only two significant digits after the decimal point
  - the overpayment, i.e., the difference between the total payment over the years and the mortgage amount, with only two significant digits after the decimal point
  - the overpayment as a percentage (in percent notation) of the mortgage amount