



Object-Based Programming

Objectives

- To understand encapsulation and data hiding.
- To understand the notions of data abstraction and abstract data types (ADTs).
- To create Java ADTs, namely, classes.
- To be able to create, use and destroy objects.
- To be able to control access to object instance variables and methods.
- To appreciate the value of object orientation.
- To understand the use of the **this** reference.
- To understand class variables and class methods.

My object all sublime

I shall achieve in time.

W. S. Gilbert

Is it a world to hide virtues in?

William Shakespeare, Twelfth Night

Your public servants serve you right.

Adlai Stevenson

But what, to serve our private ends,

Forbids the cheating of our friends?

Charles Churchill

This above all: to thine own self be true.

William Shakespeare, Hamlet

Have no friends not equal to yourself.

Confucius



Outline

- 8.1 Introduction**
- 8.2 Implementing a Time Abstract Data Type with a Class**
- 8.3 Class Scope**
- 8.4 Controlling Access to Members**
- 8.5 Creating Packages**
- 8.6 Initializing Class Objects: Constructors**
- 8.7 Using Overloaded Constructors**
- 8.8 Using Set and Get Methods**
 - 8.8.1 Executing an Applet that Uses Programmer-Defined Packages**
- 8.9 Software Reusability**
- 8.10 Final Instance Variables**
- 8.11 Composition: Objects as Instance Variables of Other Classes**
- 8.12 Package Access**
- 8.13 Using the `this` Reference**
- 8.14 Finalizers**
- 8.15 Static Class Members**
- 8.16 Data Abstraction and Encapsulation**
 - 8.16.1 Example: Queue Abstract Data Type**
- 8.17 (Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulation**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

8.1 Introduction

Now we investigate object orientation in Java in greater depth. Why did we defer this until now? First, the objects we will build will be composed in part of structured program pieces, so we needed to establish a basis in structured programming with control structures. Second, we wanted to study methods in depth. Third, we wanted to familiarize the reader with arrays that are Java objects.

Through our discussions of object-oriented Java programs in Chapter 2 through Chapter 7, we introduced many basic concepts (i.e., “object think”) and terminology (i.e., “object speak”) of object-oriented programming in Java. We also discussed our program-development methodology: We analyzed many typical problems that required a program—either a Java applet or a Java application—to be built, determined what classes from the Java API were needed to implement the program, determined what instance variables were needed, determined what methods were needed and specified how an object of our class collaborated with objects of Java API classes to accomplish the overall goals of the program.

Let us briefly review some key concepts and terminology of object orientation. OOP *encapsulates* data (*attributes*) and methods (*behaviors*) into *objects*; the data and methods of an object are intimately tied together. Objects have the property of *information hiding*. This means that although objects might know how to communicate with one another across well-defined *interfaces*, objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. Surely it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

In C and other *procedural programming languages*, programming tends to be *action-oriented*. In Java, programming is *object-oriented*. In C, the unit of programming is the *function* (called *methods* in Java). In Java, the unit of programming is the *class* from which objects are eventually *instantiated* (i.e., created). Functions do not disappear in Java; rather, they are encapsulated as methods with the data they process within the “walls” of classes.

C programmers concentrate on writing functions. Groups of actions that perform some task are formed into functions, and functions are grouped to form programs. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform. The *verbs* in a system-requirements document help the C programmer determine the set of functions that will work together to implement the system.

Java programmers concentrate on creating their own *user-defined types* called *classes*. Classes are also referred to as *programmer-defined types*. Each class contains data as well as the set of methods that manipulate the data. The data components of a class are called *instance variables* (these are called *data members* in C++). Just as an instance of a built-in type such as **int** is called a *variable*, an instance of a user-defined type (i.e., a class) is called an *object*. The focus of attention in Java is on objects rather than methods. The *nouns* in a system-requirements document help the Java programmer determine an initial set of classes with which to begin the design process. These classes are then used to instantiate objects that will work together to implement the system.

This chapter explains how to create and use objects, a subject called *object-based programming* (OBP). Chapter 9 introduces *inheritance* and *polymorphism*—two key technologies that enable true *object-oriented programming* (OOP). Although inheritance is not discussed in detail until Chapter 9, inheritance is part of every Java class definition.

Performance Tip 8.1



When passing an object to a method in Java, only a reference to the object is passed, not a copy of a possibly large object (as would be the case in a pass by value).

Software Engineering Observation 8.1



It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate their code's being modified. As we will see, classes facilitate program modifiability.

8.2 Implementing a Time Abstract Data Type with a Class

The next example consists of two classes—**Time1** (Fig. 8.1) and **TimeTest** (Fig. 8.2). Class **Time1** is defined in file **Time1.java**. Class **TimeTest** is defined in a separate file called **TimeTest.java**. It is important to note that these classes *must* be defined in

separate files, because they are both **public** classes. [Note: The output of this program appears in Fig. 8.2.]

```

1  // Fig. 8.1: Time1.java
2  // Time1 class definition maintains the time in 24-hour format.
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  public class Time1 extends Object {
8      private int hour;        // 0 - 23
9      private int minute;     // 0 - 59
10     private int second;     // 0 - 59
11
12     // Time1 constructor initializes each instance variable
13     // to zero. Ensures that each Time1 object starts in a
14     // consistent state.
15     public Time1()
16     {
17         setTime( 0, 0, 0 );
18     }
19
20     // Set a new time value using universal time. Perform
21     // validity checks on the data. Set invalid values to zero.
22     public void setTime( int h, int m, int s )
23     {
24         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
25         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
26         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
27     }
28
29     // convert to String in universal-time format
30     public String toUniversalString()
31     {
32         DecimalFormat twoDigits = new DecimalFormat( "00" );
33
34         return twoDigits.format( hour ) + ":" +
35             twoDigits.format( minute ) + ":" +
36             twoDigits.format( second );
37     }
38
39     // convert to String in standard-time format
40     public String toString()
41     {
42         DecimalFormat twoDigits = new DecimalFormat( "00" );
43
44         return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
45             ":" + twoDigits.format( minute ) +
46             ":" + twoDigits.format( second ) +
47             ( hour < 12 ? " AM" : " PM" );
48     }
49
50 } // end class Time1

```

Fig. 8.1 Abstract data type **Time1** implementation as a class.



Software Engineering Observation 8.2

Class definitions that begin with keyword **public** must be stored in a file that has exactly the same name as the class and ends with the **.java** file name extension.



Common Programming Error 8.1

Defining more than one **public** class in the same file is a syntax error.

Figure 8.1 contains a simple definition for class **Time1**. Our **Time1** class definition begins with line 6, which indicates that class **Time1** **extends** class **Object** (from package **java.lang**). Remember that you never create a class definition “from scratch.” In fact, when you create a class definition, you always use pieces of an existing class definition. Java uses *inheritance* to create new classes from existing class definitions. Keyword **extends** followed by class name **Object** indicates the class (in this case **Time1**) from which our new class inherits existing pieces. In this inheritance relationship, **Object** is called the *superclass* or *base class* and **Time1** is called the *subclass* or *derived class*. Using inheritance results in a new class definition that has the *attributes* (data) and *behaviors* (methods) of class **Object** as well as new features we add in our **Time1** class definition. Every class in Java is a subclass of **Object** (directly or indirectly). Therefore, every class inherits the 11 methods defined by class **Object**. One key **Object** method is **toString**, discussed later in this section. Other methods of class **Object** are discussed as they are needed throughout the text. For a complete list of class **Object**’s methods, see the online API documentation at

java.sun.com/j2se/1.3/docs/api/index.html



Software Engineering Observation 8.3

Every class defined in Java must extend another class. If a class does not explicitly use the keyword **extends** in its definition, the class implicitly **extends Object**.

The *body* of the class definition is delineated with left and right braces (**{** and **}**) on lines 7 and 50. Class **Time1** contains three integer instance variables—**hour**, **minute** and **second**—that represent the time in *universal-time* format (24-hour clock format).

Keywords **public** and **private** are *member access modifiers*. Instance variables or methods declared with member access modifier **public** are accessible wherever the program has a reference to a **Time1** object. Instance variables or methods declared with member access modifier **private** are accessible *only* to methods of the class in which they are defined. Every instance variable or method definition should be preceded by a member access modifier.

The three integer instance variables **hour**, **minute** and **second** are each declared (lines 8–10) with member access modifier **private**, indicating that these instance variables are accessible only to methods of the class. When a program creates (instantiates) an object of the class, such instance variables are encapsulated in the object and can be accessed only through methods of that object’s class (normally through the class’s **public** methods). Typically, instance variables are declared **private**, and methods are declared **public**. It is possible to have **private** methods and **public** data, as we will see later. The **private** methods are known as *utility methods* or *helper methods* because they can be called only by other methods of that class and are used to support the operation of those methods. Using **public** data is uncommon and is a dangerous programming practice.



Software Engineering Observation 8.4

*Methods tend to fall into a number of different categories: methods that get the values of **private** instance variables; methods that set the values of **private** instance variables; methods that implement the services of the class; and methods that perform various mechanical chores for the class, such as initializing class objects, assigning class objects, and converting between classes and built-in types or between classes and other classes.*

Access methods can read or display data. Another common use for access methods is to test whether a condition is true or false—such methods are often called *predicate methods*. An example of a predicate method would be an **isEmpty** method for any container class—a class capable of holding many objects—such as a linked list, a stack or a queue (these data structures are discussed in depth in Chapter 19, Chapter 20 and Chapter 21). A program might test **isEmpty** before attempting to read another item from the container object. A program might test **isFull** before attempting to insert another item into the container object.

Class **Time1** contains the following **public** methods—**Time1** (lines 15–18), **setTime** (lines 22–27), **toUniversalString** (lines 30–37) and **toString** (line 40–48). These are the **public methods**, **public services** or **public interface** of the class. These methods are used by *clients* (i.e., portions of a program that are users of a class) of the class to manipulate the data stored in objects of the class.

The clients of a class use references to interact with an object of the class. For example, method **paint** in an applet is a client of class **Graphics**. Method **paint** uses a reference to a **Graphics** object (such as **g**) that it receives as an argument to draw on the applet by calling methods that are **public services** of class **Graphics** (such as **drawString**, **drawLine**, **drawOval** and **drawRect**).

Notice the method with the same name as the class (lines 15–18); it is the *constructor* method of that class. A constructor is a special method that initializes the instance variables of a class object. Java calls a class's constructor method when a program instantiates an object of that class. In this example, the constructor simply calls the class's **setTime** method (discussed shortly) with hour, minute and second values specified as 0.

It is common to have several constructors for a class; this is accomplished through *method overloading* (as we will see Fig. 8.6). Constructors can take arguments but cannot specify a return data type. Implicitly, the constructor returns a reference to the instantiated object. An important difference between constructors and other methods is that constructors are *not allowed to specify a return data type* (not even **void**). Normally, constructors are **public** methods of a class. **Nonpublic** methods are discussed later.



Common Programming Error 8.2

*Attempting to declare a return type for a constructor and/or attempting to **return** a value from a constructor is a logic error. Java allows other methods of the class to have the same name as the class and to specify return types. Such methods are not constructors and will not be called when an object of the class is instantiated.*

Method **setTime** (lines 22–27) is a **public** method that receives three integer arguments and uses them to set the time. Each argument is tested in a conditional expression that determines whether the value is in range. For example, the **hour** value must be greater than or equal to 0 and less than 24 because we represent the time in universal time format (0–23 for the hour, 0–59 for the minute and 0–59 for the second). Any value outside this

range is an invalid value and is set to zero—ensuring that a **Time1** object always contains valid data. This is also known as *keeping the object in a consistent state* or *maintaining the object's integrity*. In cases where invalid data is supplied to **setTime**, the program may want to indicate that an invalid time setting was attempted. We explore this possibility in the exercises.



Good Programming Practice 8.1

Always define a class so its instance variables are maintained in a consistent state.

Method **toUniversalString** (lines 30–37) takes no arguments and returns a **String**. This method produces a universal-time-format string consisting of six digits—two for the hour, two for the minute and two for the second. For example, 13:30:07 represents 1:30:07 PM. Line 32 creates an instance of class **DecimalFormat** (from package **java.text** imported at line 3) to help format the universal time. Object **twoDigits** is initialized with the *format control string* "00", which indicates that the number format should consist of two digits—each 0 is a placeholder for a digit. If the number being formatted is a single digit, it is automatically preceded by a leading 0 (i.e., 8 is formatted as 08). The **return** statement at lines 34–36 uses method **format** (which returns a formatted **String** containing the number) from object **twoDigits** to format the **hour**, **minute** and **second** values into two-digit strings. Those strings are concatenated with the **+** operator (separated by colons) and returned from method **toUniversalString**.

Method **toString** (line 40–48) takes no arguments and returns a **String**. This method produces a standard-time-format string consisting of the **hour**, **minute** and **second** values separated by colons and an AM or PM indicator, as in 1:27:06 PM. This method uses the same **DecimalFormat** techniques as method **toUniversalString** to guarantee that the **minute** and **second** values each appear with two digits. Method **toString** is special, in that we inherited from class **Object** a **toString** method with exactly the same first line as our **toString** on line 40. The original **toString** method of class **Object** is a generic version that is used mainly as a placeholder that can be redefined by a subclass (similar to methods **init**, **start** and **paint** from class **JApplet**). Our version replaces the version we inherited to provide a **toString** method that is more appropriate for our class. This is known as *overriding* the original method definition (discussed in detail in Chapter 9).

Once the class has been defined, it can be used as a type in declarations such as

```
Time1 sunset,           // reference to object of type Time1
    timeArray[];        // reference to array of Time1 objects
```

The class name is a new type specifier. There may be many objects of a class, just as there may be many variables of a primitive data type such as **int**. The programmer can create new class types as needed; this is one of the reasons that Java is known as an *extensible language*.

The **TimeTest1** application of Fig. 8.2 uses class **Time1**. Method **main** of class **TimeTest1** declares and initializes an instance of class **Time1** called **time** in line 12. When the object is instantiated, operator **new** allocates the memory in which the **Time1** object will be stored; then **new** calls the **Time1** constructor to initialize the instance variables of the new **Time1** object. The constructor invokes method **setTime** to explicitly initialize each private instance variable to 0. Operator **new** then returns a reference to the

new object, and that reference is assigned to **time**. Similarly, line 32 in class **Time1** (Fig. 8.1) uses **new** to allocate the memory for a **DecimalFormat** object, then calls the **DecimalFormat** constructor with the argument **"00"** to indicate the number format control string.



Software Engineering Observation 8.5

*Every time **new** creates an object of a class, that class's constructor is called to initialize the instance variables of the new object.*

Note that class **Time1** was not **imported** into the **TimeTest1.java** file. Actually, every class in Java is part of a *package* (like the classes from the Java API). If the programmer does not specify the package for a class, the class is automatically placed in the *default package*, which includes the compiled classes in the current directory. If a class is in the same package as the class that uses it, an **import** statement is not required. We import classes from the Java API because their **.class** files are not in the same package with each program we write. Section 8.5 illustrates how to define your own packages of classes for reuse.

```

1  // Fig. 8.2: TimeTest1.java
2  // Class TimeTest1 to exercise class Time1
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class TimeTest1 {
8
9      // create Time1 object and manipulate it
10     public static void main( String args[] )
11     {
12         Time1 time = new Time1(); // calls Time1 constructor
13
14         // append String version of time to String output
15         String output = "The initial universal time is: " +
16             time.toUniversalString() +
17             "\nThe initial standard time is: " + time.toString() +
18             "\nImplicit toString() call: " + time;
19
20         // change time and append String version of time to output
21         time.setTime( 13, 27, 6 );
22         output += "\n\nUniversal time after setTime is: " +
23             time.toUniversalString() +
24             "\nStandard time after setTime is: " + time.toString();
25
26         // use invalid values to change time and append String
27         // version of time to output
28         time.setTime( 99, 99, 99 );
29         output += "\n\nAfter attempting invalid settings: " +
30             "\nUniversal time: " + time.toUniversalString() +
31             "\nStandard time: " + time.toString();
32     }

```

Fig. 8.2 Using an object of class **Time1** in a program (part 1 of 2).

```

33     JOptionPane.showMessageDialog( null, output,
34         "Testing Class Time1",
35         JOptionPane.INFORMATION_MESSAGE );
36
37     System.exit( 0 );
38 }
39
40 } // end class TimeTest1

```

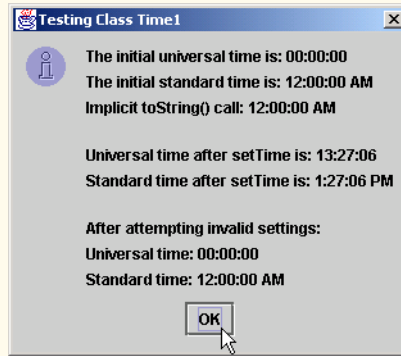


Fig. 8.2 Using an object of class **Time1** in a program (part 2 of 2).

The statement at lines 15–18 defines a **String** reference **output**, which stores the string containing the results that will be displayed in a message box. Initially, the program assigns to **output** the time in universal-time format (by sending message **toUniversalString** to the object to which **time** refers) and standard-time format (by sending message **toString** to the object to which **time** refers) to confirm that the data were initialized properly. Note that line 18 uses a special string concatenation feature of Java. Concatenating a **String** with any object results in an implicit call to the object's **toString** method to convert the object to a **String**; then the **Strings** are concatenated. Lines 17–18 illustrate that you can call **toString** both explicitly and implicitly in a **String** concatenation operation.

Line 21 sends the **setTime** message to the object to which **time** refers to change the time. Then lines 22–24 append the time to **output** again in both formats to confirm that the time was set correctly.

To illustrate that method **setTime** validates the values passed to it, line 28 calls method **setTime** and attempts to set the instance variables to invalid values. Then lines 29–31 append the time to **output** again in both formats to confirm that **setTime** validated the data. Lines 33–35 display a message box with the results of our program. Notice in the last two lines of the output window that the time is set to midnight—the default value of a **Time1** object.

Now that we have seen our first non-applet, non-application class, let us consider several issues of class design.

Again, note that the instance variables **hour**, **minute** and **second** are each declared **private**. Instance variables declared **private** are not accessible outside the class in which they are defined. The philosophy here is that the actual data representation used within the class is of no concern to the class's clients. For example, it would be perfectly

reasonable for the class to represent the time internally as the number of seconds since midnight. Clients could use the same **public** methods and get the same results without being aware of this. In this sense, implementation of a class is said to be *hidden* from its clients. Exercise 8.18 asks you to make precisely this modification to the **Time1** class of Figure 8.1 and show that there is no change visible to the clients of the class.



Software Engineering Observation 8.6

Information hiding promotes program modifiability and simplifies the client's perception of a class. The client should not require knowledge of a class's implementation (known as implementation knowledge) to be able to reuse a class.



Software Engineering Observation 8.7

Clients of a class can (and should) use the class without knowing the internal details of how the class is implemented. If the class implementation is changed (to improve performance, for example), the class clients' source code need not change, provided that the class's interface remains constant. This makes it much easier to modify systems.

In this program, the **Time1** constructor simply initializes the instance variables to 0 (i.e., the universal time equivalent of 12 AM). This ensures that the object is created in a *consistent state* (i.e., all instance variable values are valid). Invalid values cannot be stored in the instance variables of a **Time1** object, because the constructor is automatically called when the **Time1** object is created and subsequent attempts by a client to modify the instance variables are scrutinized by the method **setTime**.

Instance variables can be initialized where they are declared in the class body, by the class's constructor, or they can be assigned values by "set" methods. (Remember, instance variables that are not initialized explicitly receive default values (primitive numeric variables are set to 0, **booleans** are set to **false** and references are set to **null**).



Good Programming Practice 8.2

Initialize instance variables of a class in that class's constructor.

Every class may include a *finalizer* method called **finalize** that does "termination housekeeping" on each class object before the memory for the object is garbage collected by the system. We will discuss garbage collection and finalizers in detail in Section 8.14 and Section 8.15.

It is interesting that the **toUniversalString** and **toString** methods take no arguments. This is because these methods implicitly know that they are to manipulate the instance variables of the particular **Time1** object for which they are invoked. This makes method calls more concise than conventional function calls in procedural programming. It also reduces the likelihood of passing the wrong arguments, the wrong types of arguments and/or the wrong number of arguments, as often happens in C function calls.



Software Engineering Observation 8.8

Using an object-oriented programming approach can often simplify method calls by reducing the number of parameters to be passed. This benefit of object-oriented programming derives from the fact that encapsulation of instance variables and methods within an object gives the methods the right to access the instance variables.

Classes simplify programming because the client (or user of the class object) need only be concerned with the **public** operations encapsulated in the object. Such operations are

usually designed to be client oriented rather than implementation oriented. Clients need not be concerned with a class's implementation. Interfaces do change, but less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. By hiding the implementation, we eliminate the possibility of other program parts becoming dependent on the details of the class implementation.

Often, classes do not have to be created “from scratch.” Rather, they can be *derived* from other classes that provide operations the new classes can use, or classes can include objects of other classes as members. Such *software reuse* can greatly enhance programmer productivity. Deriving new classes from existing classes is called *inheritance*—a distinguishing feature between object-based programming and object-oriented programming—and is discussed in detail in Chapter 9. Including class objects as members of other classes is called *composition* or *aggregation* and is discussed later in this chapter.

8.3 Class Scope

A class's instance variables and methods belong to that *class's scope*. Within a class's scope, class members are accessible to all of that class's methods and can be referenced simply by name. Outside a class's scope, class members cannot be referenced directly by name. Those class members (such as **public** members) that are visible can be accessed only through a “handle” (i.e., primitive data type members can be referred to by *object-ReferenceName.primitiveVariableName* and object members can be referenced by *object-ReferenceName.objectMemberName*). For example, a program can determine the number of elements in an array object by accessing the array's **public** member **length** as in *arrayName.length*.

Variables defined in a method are known only to that method (i.e., they are local variables to that method). Such variables are said to have block scope. If a method defines a variable with the same name as a variable with class scope (i.e., an instance variable), the class-scope variable is hidden by the method-scope variable in the method scope. A hidden instance variable can be accessed in the method by preceding its name with the keyword **this** and the dot operator, as in **this.variableName**. Keyword **this** is discussed Section 8.13.

8.4 Controlling Access to Members

The member access modifiers **public** and **private** control access to a class's instance variables and methods. (In Chapter 9, we will introduce the additional access modifier **protected**.) As we stated previously, the primary purpose of **public** methods is to present to the class's clients a view of the *services* the class provides (i.e., the public interface of the class). Clients of the class need not be concerned with how the class accomplishes its tasks. For this reason, the **private** instance variables and **private** methods of a class (i.e., the class's implementation details) are not accessible to the clients of a class. Restricting access to class members via keyword **private** is called *encapsulation*.



Common Programming Error 8.3

An attempt by a method that is not a member of a particular class to access a **private** member of that class is a syntax error.

Figure 8.3 demonstrates that **private** class members are not accessible by name outside the class. Line 10 attempts to access directly the **private** instance variable **hour** of

the **Time1** object to which **time** refers. When this program is compiled, the compiler generates an error stating that the **private** member **hour** is not accessible. [Note: This program assumes that the **Time1** class from Figure 8.1 is used.]



Good Programming Practice 8.3

Our preference is to list the **private** instance variables of a class first, so that, as you read the code, you see the names and types of the instance variables before they are used in the methods of the class.



Software Engineering Observation 8.9

Keep all the instance variables of a class **private**. When necessary, provide **public** methods to set the values of **private** instance variables and to get the values of **private** instance variables. This architecture helps hide the implementation of a class from its clients, which reduces bugs and improves program modifiability.

Access to **private** data should be controlled carefully by the class's methods. For example, to allow clients to read the value of **private** data, the class can provide a “*get*” method (also called an *accessor method*). To enable clients to modify **private** data, the class can provide a “*set*” method (also called a *mutator method*). Such modification would seem to violate the notion of **private** data, but a *set* method can provide data validation capabilities (such as range checking) to ensure that the value is set properly. A *set* method can also translate between the form of the data used in the interface and the form used in the implementation. A *get* method need not expose the data in “raw” format; rather, the *get* method can edit the data and limit the view of the data the client will see.



Software Engineering Observation 8.10

Class designers use **private** data and **public** methods to enforce the notion of information hiding and the principle of least privilege. If the client of a class needs access to data in the class, provide that access through **public** methods of the class. By doing so, the programmer of the class controls how the class's data is manipulated (e.g., data validity checking can prevent invalid data from being stored in an object). This is the principle of data encapsulation.

```

1  // Fig. 8.3: TimeTest2.java
2  // Demonstrate errors resulting from attempts
3  // to access private members of class Time1.
4  public class TimeTest2 {
5
6      public static void main( String args[] )
7      {
8          Time1 time = new Time1();
9
10         time.hour = 7;
11     }
12 }
```

```

TimeTest2.java:10: hour has private access in Time1
    time.hour = 7;
        ^
1 error
```

Fig. 8.3 Erroneous attempt to access private members of class **Time1**.



Software Engineering Observation 8.11

The class designer need not provide set and/or get methods for each **private** data member; these capabilities should be provided only when it makes sense and after careful thought by the class designer.



Testing and Debugging Tip 8.1

Making the instance variables of a class **private** and the methods of the class **public** facilitates debugging because problems with data manipulations are localized to the class's methods.

8.5 Creating Packages

As we have seen in almost every example in the text, classes and *interfaces* (discussed in Chapter 9) from preexisting libraries, such as the Java API, can be imported into a Java program. Each class and interface in the Java API belongs to a specific package that contains a group of related classes and interfaces. As applications become more complex, packages help programmers manage the complexity of application components. Packages also facilitate software reuse by enabling programs to import classes from other packages (as we have done in almost every example to this point). Another benefit of packages is that they provide a convention for *unique class names*. With hundreds of thousands of Java programmers around the world, there is a good chance that the names you choose for classes will conflict with the names that other programmers choose for their classes. This section introduces how to create your own packages and discusses the standard distribution mechanism for packages.

The application of Fig. 8.4 and Fig. 8.5 illustrates how to create your own package and use a class from that package in a program. The steps for creating a reusable class are:

1. Define a **public** class. If the class is not **public**, it can be used only by other classes in the same package.
2. Choose a package name, and add a **package** statement to the source code file for the reusable class definition. [Note: There can be only one **package** statement in a Java source code file.]
3. Compile the class so it is placed in the appropriate package directory structure.
4. Import the reusable class into a program, and use the class.



Common Programming Error 8.4

A syntax error occurs if any code appears in a Java file before the **package** statement (if there is one) in the file.

We chose to demonstrate *Step 1* by modifying the **public** class **Time1** defined in Fig. 8.1. The new version is shown in Fig. 8.4. No modifications have been made to the implementation of the class, so we will not discuss the implementation details of the class again here.

To satisfy *Step 2*, we added a **package** statement at the beginning of the file. Line 3 uses a **package** statement to define a **package** named **com.deitel.jhttp4.ch08**. Placing a **package** statement at the beginning of a Java source file indicates that the class defined in the file is part of the specified package. The only types of statements in Java that can appear outside the braces of a class definition are **package** statements and **import** statements.

```

1  // Fig. 8.4: Time1.java
2  // Time1 class definition in a package
3  package com.deitel.jhtp4.ch08;
4
5  // Java core packages
6  import java.text.DecimalFormat;
7
8  public class Time1 extends Object {
9      private int hour;        // 0 - 23
10     private int minute;     // 0 - 59
11     private int second;     // 0 - 59
12
13     // Time1 constructor initializes each instance variable
14     // to zero. Ensures that each Time1 object starts in a
15     // consistent state.
16     public Time1()
17     {
18         setTime( 0, 0, 0 );
19     }
20
21     // Set a new time value using universal time. Perform
22     // validity checks on the data. Set invalid values to zero.
23     public void setTime( int h, int m, int s )
24     {
25         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
26         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
27         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
28     }
29
30     // convert to String in universal-time format
31     public String toUniversalString()
32     {
33         DecimalFormat twoDigits = new DecimalFormat( "00" );
34
35         return twoDigits.format( hour ) + ":" +
36             twoDigits.format( minute ) + ":" +
37             twoDigits.format( second );
38     }
39
40     // convert to String in standard-time format
41     public String toString()
42     {
43         DecimalFormat twoDigits = new DecimalFormat( "00" );
44
45         return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
46             ":" + twoDigits.format( minute ) +
47             ":" + twoDigits.format( second ) +
48             ( hour < 12 ? " AM" : " PM" );
49     }
50
51 } // end class Time1

```

Fig. 8.4 Placing Class **Time1** in a package for reuse.



Software Engineering Observation 8.12

*A Java source code file has the following order: a **package** statement (if any), zero or more **import** statements, then class definitions. Only one of the class definitions in a particular file can be **public**. Other classes in the file are also placed in the package, but are reusable only from other classes in that package—they cannot be imported into classes in another package. They are in the package to support the reusable class in the file.*

In an effort to provide unique names for every package, Sun Microsystems specifies a convention for package naming that all Java programmers should follow. Every package name should start with your Internet domain name in reverse order. For example, our Internet domain name is **deitel.com**, so we began our package name with **com.deitel**. If your domain name is **yourcollege.edu**, the package name you would use is **edu.yourcollege**. After the domain name is reversed, you can choose any other names you want for your package. If you are part of a company with many divisions or a university with many schools, you may want to use the name of your division or school as the next name in the package. We chose to use **jhtp4** as the next name in our package name to indicate that this class is from *Java How to Program: Fourth Edition*. The last name in our package name specifies that this package is for Chapter 8 (**ch08**). [Note: We use our own packages several times throughout the book. You can determine the chapter in which one of our reusable classes is defined by looking at the last part of the package name in the **import** statement. This appears before the name of the class being imported or before the ***** if a particular class is not specified.]

Step 3 is to compile the class so it is stored in the appropriate package. When a Java file containing a **package** statement is compiled, the resulting **.class** file is placed in the directory structure specified by the **package** statement. The preceding **package** statement indicates that class **Time1** should be placed in the directory **ch08**. The other names—**com**, **deitel** and **jhtp4**—are also directories. The directory names in the **package** statement specify the exact location of the classes in the package. If these directories do not exist before the class is compiled, the compiler creates them.

When compiling a class in a package, there is an extra option (**-d**) that must be passed to the **javac** compiler. This option specifies where to create (or locate) the directories in the **package** statement. For example, we used the compilation command

```
javac -d . Time1.java
```

to specify that the first directory specified in our package name should be placed in the current directory. The **.** after **-d** in the preceding command represents the current directory on the Windows, UNIX and Linux operating systems (and several others as well). After executing the compilation command, the current directory contains a directory called **com**, **com** contains a directory called **deitel**, **deitel** contains a directory called **jhtp4** and **jhtp4** contains a directory called **ch08**. In the **ch08** directory, you can find the file **Time1.class**.

The **package** directory names become part of the class name when the class is compiled. The class name in this example is actually **com.deitel.jhtp4.ch08.Time1** after the class is compiled. You can use this *fully qualified* name in your programs or you can **import** the class and use its short name (**Time1**) in the program. If another package also contains a **Time1** class, the fully qualified class names can be used to distinguish between the classes in the program and prevent a naming conflict (also called a name collision).

Once the class is compiled and stored in its package, the class can be imported into programs (Step 4). The **TimeTest3** application of Fig. 8.5, line 8 specifies that class **Time1** should be **imported** for use in class **TimeTest3**.

At compile time for class **TimeTest3**, **javac** must locate **.class** file for **Time1**, so **javac** can ensure that class **TimeTest3** uses class **Time1** correctly. The compiler follows a specific search order to locate the classes it needs. It begins by searching the standard Java classes that are bundled with the J2SDK. Then it searches for *extension classes*. Java 2 provides an *extensions mechanism* that enables new packages to be added to Java for development and execution purposes. [Note: The extensions mechanism is beyond the scope of this book. For more information, visit java.sun.com/j2se/1.3/docs/guide/extensions.] If the class is not found in the standard Java classes or in the extension classes, the compiler searches the *class path*. By default, the class path consists only of the current directory. However, the class path can be modified by:

```

1  // Fig. 8.5: TimeTest3.java
2  // Class TimeTest3 to use imported class Time1
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  // Deitel packages
8  import com.deitel.jhtp4.ch08.Time1; // import Time1 class
9
10 public class TimeTest3 {
11
12     // create an object of class Time1 and manipulate it
13     public static void main( String args[] )
14     {
15         Time1 time = new Time1(); // create Time1 object
16
17         time.setTime( 13, 27, 06 ); // set new time
18         String output =
19             "Universal time is: " + time.toUniversalString() +
20             "\nStandard time is: " + time.toString();
21
22         JOptionPane.showMessageDialog( null, output,
23             "Packaging Class Time1 for Reuse",
24             JOptionPane.INFORMATION_MESSAGE );
25
26         System.exit( 0 );
27     }
28
29 } // end class TimeTest3

```

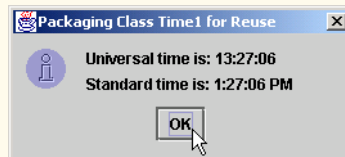


Fig. 8.5 Using programmer-defined class **Time1** in a package.

1. providing the **-classpath** option to the **javac** compiler, or
2. setting the **CLASSPATH** environment variable (a special variable that you define and the operating system maintains so that applications can search for classes in the specified locations).

In each case, the class path consists of a list of directories and/or *archive files* separated by semicolons (;). Archive files are individual files that contain directories of other files, typically in compressed format. For example, the standard classes of Java are contained in the archive file **rt.jar** that is installed with the J2SDK. Archive files normally end with the **.jar** or **.zip** file name extensions. The directories and archive files specified in the class path contain the classes you wish to make available to the Java compiler. For more information on the class path, visit java.sun.com/j2se/1.3/docs/tooldocs/win32/classpath.html for Windows or java.sun.com/j2se/1.3/docs/tooldocs/solaris/classpath.html for Solaris/Linux. [Note: We discuss archive files in more detail in Section 8.8.]



Common Programming Error 8.5

Specifying an explicit class path eliminates the current directory from the class path. This prevents classes in the current directory from loading properly. If classes must be loaded from the current directory, include the current directory (.) in the explicit class path.



Software Engineering Observation 8.13

In general, it is a better practice to use the **-classpath** option of the compiler, rather than the **CLASSPATH** environment variable, to specify the class path for a program. This enables each application to have its own class path.



Testing and Debugging Tip 8.2

Specifying the class path with the **CLASSPATH** environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same packages.

For the example of Fig. 8.4 and Fig. 8.5, we did not specify an explicit class path. Thus, to locate the classes in the **com.deitel.jhtp4.ch08** package from this example, the compiler looks in the current directory for the first name in the package—**com**. Next, the compiler navigates the directory structure. Directory **com** contains the subdirectory **deitel**. Directory **deitel** contains the subdirectory **jhtp4**. Finally, directory **jhtp4** contains subdirectory **ch08**. In the **ch08** directory is the file **Time1.class**, which is loaded by the compiler to ensure that the class is used properly in our program.

Locating the classes to execute the program is similar to locating the classes to compile the program. Like the compiler, the **java** interpreter searches the standard classes and extension classes first, then searches the class path (the current directory by default). The class path for the interpreter can be specified explicitly by using either of the techniques discussed for the compiler. As with the compiler, it is better to specify an individual program's class path via command-line options to the interpreter. You can specify the class path to the **java** interpreter via the **-classpath** or **-cp** command line options followed by a list of directories and/or archive files separated by semicolons (;).

8.6 Initializing Class Objects: Constructors

When an object is created, its members can be initialized by a *constructor* method. A constructor is a method with the same name as the class (including case sensitivity). The pro-

programmer of a class can define the class's constructor, which is invoked each time the program instantiates an object of that class. Instance variables can be initialized implicitly to their default values (**0** for primitive numeric types, **false** for **booleans** and **null** for references), they can be initialized in a constructor of the class or their values can be set later after the object is created. Constructors cannot specify return types or return values. A class can contain overloaded constructors to provide a variety of means for initializing objects of that class.

When a program instantiates an object of a class, the program can supply *initializers* in parentheses to the right of the class name. These initializers are passed as arguments to the class's constructor. This technique is demonstrated in the example of Section 8.7. We have also seen this technique several times previously as we created new objects of classes like **DecimalFormat**, **JLabel**, **TextField**, **TextArea** and **Button**. For each of these classes, we have seen statements of the form

```
ref = new ClassName( arguments );
```

where **ref** is a reference of the appropriate data type, **new** indicates that a new object is being created, *ClassName* indicates the type of the new object and *arguments* specifies the values used by the class's constructor to initialize the object.

If no constructors are defined for a class, the compiler creates a *default constructor* that takes no arguments (also called a *no-argument constructor*). The default constructor for a class calls the default constructor for its superclass (the class it extends), then proceeds to initialize the instance variables in the manner we discussed previously. If the class that this class extends does not have a default constructor, the compiler issues an error message. It is also possible for the programmer to provide a no-argument constructor, as we showed in class **Time1** and will see in the next example. If any constructors are defined for a class by the programmer, Java will not create a default constructor for the class.

Good Programming Practice 8.4



When appropriate (almost always), provide a constructor to ensure that every object is properly initialized with meaningful values.

Common Programming Error 8.6



If constructors are provided for a class, but none of the **public** constructors are no-argument constructors, and an attempt is made to call a no-argument constructor to initialize an object of the class, a syntax error occurs. A constructor can be called with no arguments only if there are no constructors for the class (the default constructor is called) or if there is a no-argument constructor.

8.7 Using Overloaded Constructors

Methods of a class can be *overloaded* (i.e., several methods in a class may have exactly the same name as defined in Chapter 6, Methods). To overload a method of a class, simply provide a separate method definition with the same name for each version of the method. Remember that overloaded methods *must* have different parameter lists.

Common Programming Error 8.7



Attempting to overload a method of a class with another method that has the exact same signature (name and parameters) is a syntax error.

The **Time1** constructor in Fig. 8.1 initialized **hour**, **minute** and **second** to **0** (i.e., 12 midnight in universal time) with a call to the class's **setTime** method. Figure 8.6

overloads the constructor method to provide a convenient variety of ways to initialize objects of the new class **Time2**. The constructors guarantee that every object begins its existence in a consistent state. In this program, each constructor calls method **setTime** with the values passed to the constructor, to ensure that the value supplied for **hour** is in the range 0 to 23 and that the values for **minute** and **second** are each in the range 0 to 59. If a value is out of range, it is set to zero by **setTime** (once again ensuring that each instance variable remains in a consistent state). The appropriate constructor is invoked by matching the number, types and order of the arguments specified in the constructor call with the number, types and order of the parameters specified in each constructor definition. The matching constructor is called automatically. Figure 8.7 uses class **Time2** to demonstrate its constructors.

```

1  // Fig. 8.6: Time2.java
2  // Time2 class definition with overloaded constructors.
3  package com.deitel.jhtp4.ch08;
4
5  // Java core packages
6  import java.text.DecimalFormat;
7
8  public class Time2 extends Object {
9      private int hour;        // 0 - 23
10     private int minute;     // 0 - 59
11     private int second;     // 0 - 59
12
13     // Time2 constructor initializes each instance variable
14     // to zero. Ensures that Time object starts in a
15     // consistent state.
16     public Time2()
17     {
18         setTime( 0, 0, 0 );
19     }
20
21     // Time2 constructor: hour supplied, minute and second
22     // defaulted to 0
23     public Time2( int h )
24     {
25         setTime( h, 0, 0 );
26     }
27
28     // Time2 constructor: hour and minute supplied, second
29     // defaulted to 0
30     public Time2( int h, int m )
31     {
32         setTime( h, m, 0 );
33     }
34
35     // Time2 constructor: hour, minute and second supplied
36     public Time2( int h, int m, int s )
37     {
38         setTime( h, m, s );
39     }

```

Fig. 8.6 Class **Time2** with overloaded constructors (part 1 of 2).

```

40
41 // Time2 constructor: another Time2 object supplied
42 public Time2( Time2 time )
43 {
44     setTime( time.hour, time.minute, time.second );
45 }
46
47 // Set a new time value using universal time. Perform
48 // validity checks on data. Set invalid values to zero.
49 public void setTime( int h, int m, int s )
50 {
51     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
52     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
53     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
54 }
55
56 // convert to String in universal-time format
57 public String toUniversalString()
58 {
59     DecimalFormat twoDigits = new DecimalFormat( "00" );
60
61     return twoDigits.format( hour ) + ":" +
62         twoDigits.format( minute ) + ":" +
63         twoDigits.format( second );
64 }
65
66 // convert to String in standard-time format
67 public String toString()
68 {
69     DecimalFormat twoDigits = new DecimalFormat( "00" );
70
71     return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
72         ":" + twoDigits.format( minute ) +
73         ":" + twoDigits.format( second ) +
74         ( hour < 12 ? " AM" : " PM" );
75 }
76
77 } // end class Time2

```

Fig. 8.6 Class **Time2** with overloaded constructors (part 2 of 2).

Most of the code in class **Time2** is identical to that in class **Time1**, so we concentrate on only the new features here (i.e., the constructors). Lines 16–19 define the no-argument (default) constructor. Lines 23–26 define a **Time2** constructor that receives a single **int** argument representing the **hour**. Lines 30–33 defines a **Time2** constructor that receives two **int** arguments representing the **hour** and **minute**. Lines 36–39 define a **Time2** constructor that receives three **int** arguments representing the **hour**, **minute** and **second**. Lines 42–45 define a **Time2** constructor that receives a **Time2** reference to another **Time2** object. In this case, the values from the **Time2** argument are used to initialize the **hour**, **minute** and **second**. Notice that none of the constructors specifies a return data type (remember, this is not allowed for constructors). Also, notice that all the constructors receive different numbers of arguments and/or different types of arguments. Even though only two

of the constructors receive values for the **hour**, **minute** and **second**, all the constructors call **setTime** with values for **hour**, **minute** and **second** and substitute zeros for the missing values to satisfy **setTime**'s requirement of three arguments.

Notice in particular the constructor at lines 42–45, which uses the **hour**, **minute** and **second** values of its argument **time** to initialize the new **Time2** object. Even though we know that **hour**, **minute** and **second** are declared as **private** variables of class **Time2**, we are able to access these values directly by using the expressions **time.hour**, **time.minute** and **time.second**. This is due to a special relationship between objects of the same class. When one object of a class has a reference to another object of the same class, the first object can access *all* the second object's data and methods.

Class **TimeTest4** (Fig. 8.7) creates six **Time2** objects (lines 17–22) to demonstrate how to invoke the different constructors of the class. Line 17 shows that the no-argument constructor is invoked by placing an empty set of parentheses after the class name when allocating a **Time2** object with **new**. Lines 18–22 of the program demonstrate passing arguments to the **Time2** constructors. Remember that the appropriate constructor is invoked by matching the number, types and order of the arguments specified in the constructor call with the number, types and order of the parameters specified in each method definition. So, line 18 invokes the constructor at line 23 of Fig. 8.6. Line 19 invokes the constructor at line 30 of Fig. 8.6. Lines 20–21 invoke the constructor at line 36 of Fig. 8.6. Line 22 invokes the constructor at line 42 of Fig. 8.6.

Note that each **Time2** constructor in Fig. 8.6 could be written to include a copy of the appropriate statements from method **setTime**. This could be slightly more efficient, because the extra call to **setTime** is eliminated. However, consider changing the representation of the time from three **int** values (requiring 12 bytes of memory) to a single **int** value representing the total number of seconds that have elapsed in the day (requiring 4 bytes of memory). Coding the **Time2** constructors and method **setTime** identically makes such a change in this class definition more difficult. If the implementation of method **setTime** changes, the implementation of the **Time2** constructors would need to change accordingly. Having the **Time2** constructors call **setTime** directly requires any changes to the implementation of **setTime** to be made only once. This reduces the likelihood of a programming error when altering the implementation.



Software Engineering Observation 8.14

If a method of a class already provides all or part of the functionality required by a constructor (or other method) of the class, call that method from the constructor (or other method). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. It is also an effective example of reuse.

```

1  // Fig. 8.7: TimeTest4.java
2  // Using overloaded constructors
3
4  // Java extension packages
5  import javax.swing.*;
6

```

Fig. 8.7 Using overloaded constructors to initialize objects of class **Time2** (part 1 of 3).

```

7  // Deitel packages
8  import com.deitel.jhtp4.ch08.Time2;
9
10 public class TimeTest4 {
11
12     // test constructors of class Time2
13     public static void main( String args[] )
14     {
15         Time2 t1, t2, t3, t4, t5, t6;
16
17         t1 = new Time2();           // 00:00:00
18         t2 = new Time2( 2 );       // 02:00:00
19         t3 = new Time2( 21, 34 );  // 21:34:00
20         t4 = new Time2( 12, 25, 42 ); // 12:25:42
21         t5 = new Time2( 27, 74, 99 ); // 00:00:00
22         t6 = new Time2( t4 );      // 12:25:42
23
24         String output = "Constructed with: " +
25             "\nt1: all arguments defaulted" +
26             "\n      " + t1.toUniversalString() +
27             "\n      " + t1.toString();
28
29         output += "\nt2: hour specified; minute and " +
30             "second defaulted" +
31             "\n      " + t2.toUniversalString() +
32             "\n      " + t2.toString();
33
34         output += "\nt3: hour and minute specified; " +
35             "second defaulted" +
36             "\n      " + t3.toUniversalString() +
37             "\n      " + t3.toString();
38
39         output += "\nt4: hour, minute, and second specified" +
40             "\n      " + t4.toUniversalString() +
41             "\n      " + t4.toString();
42
43         output += "\nt5: all invalid values specified" +
44             "\n      " + t5.toUniversalString() +
45             "\n      " + t5.toString();
46
47         output += "\nt6: Time2 object t4 specified" +
48             "\n      " + t6.toUniversalString() +
49             "\n      " + t6.toString();
50
51         JOptionPane.showMessageDialog( null, output,
52             "Demonstrating Overloaded Constructors",
53             JOptionPane.INFORMATION_MESSAGE );
54
55         System.exit( 0 );
56     }
57
58 } // end class TimeTest4

```

Fig. 8.7 Using overloaded constructors to initialize objects of class **Time2** (part 2 of 3).

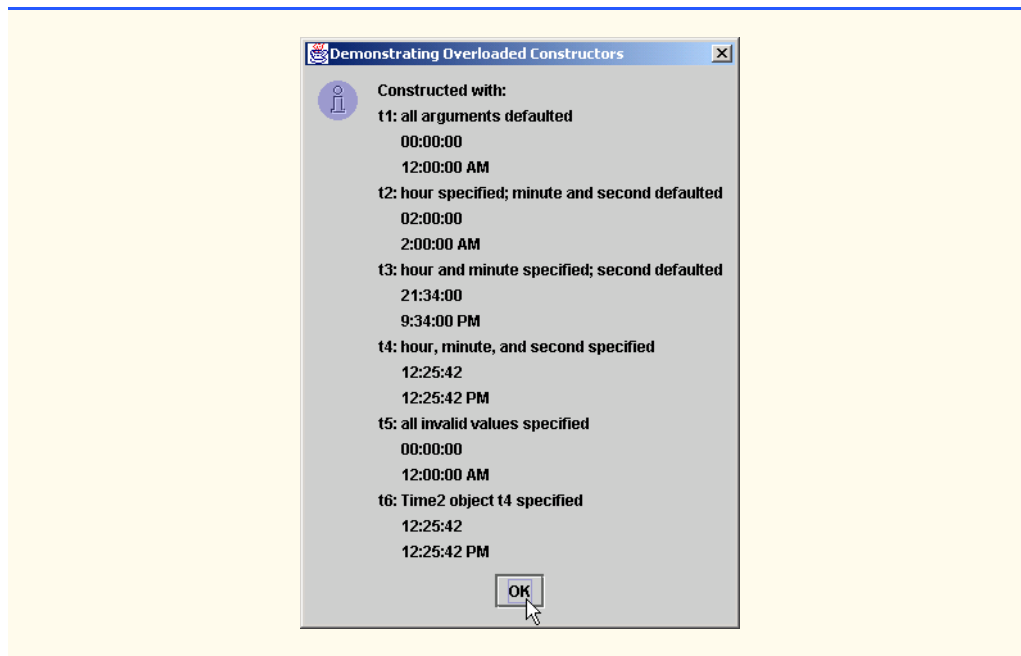


Fig. 8.7 Using overloaded constructors to initialize objects of class **Time2** (part 3 of 3).

8.8 Using Set and Get Methods

Private instance variables can be manipulated only by methods of the class. A typical manipulation might be the adjustment of a customer's bank balance (e.g., a **private** instance variable of a class **BankAccount**) by a method **computeInterest**.

Classes often provide **public** methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) **private** instance variables. These methods need not be called *set* and *get*, but they often are.

As a naming example, a method that sets instance variable **interestRate** would typically be named **setInterestRate** and a method that gets the **interestRate** would typically be called **getInterestRate**. *Get* methods are also commonly called *accessor methods* or *query methods*. *Set* methods are also commonly called *mutator methods* (because they typically change a value).

It would seem that providing *set* and *get* capabilities is essentially the same as making the instance variables **public**. This is another subtlety of Java that makes the language so desirable for software engineering. If an instance variable is **public**, the instance variable can be read or written at will by any method in the program. If an instance variable is **private**, a **public** *get* method certainly seems to allow other methods to read the data at will but the *get* method controls the formatting and display of the data. A **public** *set* method can—and most likely will—carefully scrutinize attempts to modify the instance variable's value. This ensures that the new value is appropriate for that data item. For example, an attempt to *set* the day of the month for a date to 37 would be rejected, an attempt to *set* a person's weight to a negative value would be rejected, and so on. So,

although *set* and *get* methods could provide access to **private** data, the access is restricted by the programmer's implementation of the methods.

The benefits of data integrity are not automatic simply because instance variables are made **private**—the programmer must provide validity checking. Java provides the framework in which programmers can design better programs in a convenient manner.



Software Engineering Observation 8.15

*Methods that set the values of **private** data should verify that the intended new values are proper; if they are not, the set methods should place the **private** instance variables into an appropriate consistent state.*

A class's *set* methods can return values indicating that attempts were made to assign invalid data to objects of the class. This enables clients of the class to test the return values of *set* methods to determine whether the objects they are manipulating are valid and to take appropriate action if the objects are not valid. In Chapter 14, Exception Handling, we illustrate a more robust way in which clients of a class can be notified if an object is not valid.



Good Programming Practice 8.5

Every method that modifies the private instance variables of an object should ensure that the data remains in a consistent state.

The applet of Fig. 8.8 (class **Time3**) and Fig. 8.9 (class **TimeTest5**) enhances our **Time** class (now called **Time3**) to include *get* and *set* methods for the **hour**, **minute** and **second** **private** instance variables. The *set* methods strictly control the setting of the instance variables to valid values. Attempts to set any instance variable to an incorrect value cause the instance variable to be set to zero (thus leaving the instance variable in a consistent state). Each *get* method simply returns the appropriate instance variable's value. This applet introduces enhanced GUI event handling techniques as we move toward defining our first full-fledged windowed application. After discussing the code, we introduce how to set up an applet to use classes in programmer-defined packages.

```

1  // Fig. 8.8: Time3.java
2  // Time3 class definition with set and get methods
3  package com.deitel.jhtp4.ch08;
4
5  // Java core packages
6  import java.text.DecimalFormat;
7
8  public class Time3 extends Object {
9      private int hour;        // 0 - 23
10     private int minute;     // 0 - 59
11     private int second;     // 0 - 59
12
13     // Time3 constructor initializes each instance variable
14     // to zero. Ensures that Time object starts in a
15     // consistent state.
16     public Time3()
17     {
18         setTime( 0, 0, 0 );
19     }

```

Fig. 8.8 Class **Time3** with *set* and *get* methods (part 1 of 3).

```
20
21 // Time3 constructor: hour supplied, minute and second
22 // defaulted to 0
23 public Time3( int h )
24 {
25     setTime( h, 0, 0 );
26 }
27
28 // Time3 constructor: hour and minute supplied, second
29 // defaulted to 0
30 public Time3( int h, int m )
31 {
32     setTime( h, m, 0 );
33 }
34
35 // Time3 constructor: hour, minute and second supplied
36 public Time3( int h, int m, int s )
37 {
38     setTime( h, m, s );
39 }
40
41 // Time3 constructor: another Time3 object supplied
42 public Time3( Time3 time )
43 {
44     setTime( time.getHour(), time.getMinute(),
45             time.getSecond() );
46 }
47
48 // Set Methods
49 // Set a new time value using universal time. Perform
50 // validity checks on data. Set invalid values to zero.
51 public void setTime( int h, int m, int s )
52 {
53     setHour( h ); // set the hour
54     setMinute( m ); // set the minute
55     setSecond( s ); // set the second
56 }
57
58 // validate and set hour
59 public void setHour( int h )
60 {
61     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
62 }
63
64 // validate and set minute
65 public void setMinute( int m )
66 {
67     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
68 }
69
```

Fig. 8.8 Class **Time3** with *set* and *get* methods (part 2 of 3).

```

70 // validate and set second
71 public void setSecond( int s )
72 {
73     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
74 }
75
76 // Get Methods
77 // get hour value
78 public int getHour()
79 {
80     return hour;
81 }
82
83 // get minute value
84 public int getMinute()
85 {
86     return minute;
87 }
88
89 // get second value
90 public int getSecond()
91 {
92     return second;
93 }
94
95 // convert to String in universal-time format
96 public String toUniversalString()
97 {
98     DecimalFormat twoDigits = new DecimalFormat( "00" );
99
100     return twoDigits.format( getHour() ) + ":" +
101         twoDigits.format( getMinute() ) + ":" +
102         twoDigits.format( getSecond() );
103 }
104
105 // convert to String in standard-time format
106 public String toString()
107 {
108     DecimalFormat twoDigits = new DecimalFormat( "00" );
109
110     return ( ( getHour() == 12 || getHour() == 0 ) ?
111         12 : getHour() % 12 ) + ":" +
112         twoDigits.format( getMinute() ) + ":" +
113         twoDigits.format( getSecond() ) +
114         ( getHour() < 12 ? " AM" : " PM" );
115 }
116
117 } // end class Time3

```

Fig. 8.8 Class **Time3** with *set* and *get* methods (part 3 of 3).

The new *set* methods of the class are defined in Fig. 8.8 at lines 59–62, 65–68 and 71–74, respectively. Notice that each method performs the same conditional statement that was

previously in method **setTime** for setting the **hour**, **minute** or **second**. The addition of these methods caused us to redefine the body of method **setTime** to follow *Software Engineering Observation 8.14*—if a method of a class already provides all or part of the functionality required by another method of the class, call that method from the other method. Notice that **setTime** (lines 51–56) now calls methods **setHour**, **setMinute** and **setSecond**—each of which performs part of **setTime**’s task.

The new *get* methods of the class are defined at lines 78–81, 84–87 and 90–93, respectively. Notice that each method simply returns the **hour**, **minute** or **second** value (a copy of each value is returned, because these are all primitive data type variables). The addition of these methods caused us to redefine the bodies of methods **toUniversalString** (lines 96–103) and **toString** (lines 106–115) to follow *Software Engineering Observation 8.14*. In both cases, every use of instance variables **hour**, **minute** and **second** is replaced with a call to **getHour**, **getMinute** and **getSecond**.

Due to the changes in class **Time3** just described, we have minimized the changes that will have to occur in the class definition if the data representation is changed from **hour**, **minute** and **second** to another representation (such as total elapsed seconds in the day). Only the new *set* and *get* method bodies will have to change. This allows the programmer to change the implementation of the class without affecting the clients of the class (as long as all the **public** methods of the class are still called the same way).

The **TimeTest5** applet (Fig. 8.9) provides a graphical user interface that enables the user to exercise the methods of class **Time3**. The user can set the hour, minute or second value by typing a value in the appropriate **JTextField** and pressing the *Enter* key. The user can also click the “Add 1 to second” button to increment the time by one second. The **JTextField** and **JButton** events in this applet are all processed in method **actionPerformed** (lines 66–94). Notice that lines 34, 41, 48 and 59 all call **addActionListener** to indicate that the applet should start listening to **JTextFields** **hourField**, **minuteField**, **secondField** and **JButton** **tickButton**, respectively. Also, notice that all four calls use **this** as the argument, indicating that the object of our applet class **TimeTest5** has its **actionPerformed** method invoked for each user interaction with these four GUI components. This poses an interesting question—how do we determine the GUI component with which the user interacted?

In **actionPerformed**, notice the use of **actionEvent.getSource()** to determine which GUI component generated the event. For example, line 69 determines whether **tickButton** was clicked by the user. If so, the body of the **if** structure executes. Otherwise, the program tests the condition in the **if** structure at line 73, and so on. Every event has a *source*—the GUI component with which the user interacted to signal the program to do a task. The **ActionEvent** parameter contains a reference to the source of the event. The condition in line 69 simply asks, “Is the *source* of the event the **tickButton**?” This condition compares the references on either side of the **==** operator to determine whether they refer to the same object. In this case, if they both refer to the **JButton**, then the program knows that the user pressed the button. Remember that the source of the event calls **actionPerformed** in response to the user interaction.

After each operation, the resulting time is displayed as a string in the status bar of the applet. The output windows in Fig. 8.9 illustrate the applet before and after the following operations: setting the hour to 23, setting the minute to 59, setting the second to 58 and incrementing the second twice with the “Add 1 to second” button.

```
1  // Fig. 8.9: TimeTest5.java
2  // Demonstrating the Time3 class set and get methods.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 // Deitel packages
12 import com.deitel.jhtp4.ch08.Time3;
13
14 public class TimeTest5 extends JApplet
15     implements ActionListener {
16
17     private Time3 time;
18     private JLabel hourLabel, minuteLabel, secondLabel;
19     private JTextField hourField, minuteField,
20         secondField, displayField;
21     private JButton tickButton;
22
23     // Create Time3 object and set up GUI
24     public void init()
25     {
26         time = new Time3();
27
28         Container container = getContentPane();
29         container.setLayout( new FlowLayout() );
30
31         // set up hourLabel and hourField
32         hourLabel = new JLabel( "Set Hour" );
33         hourField = new JTextField( 10 );
34         hourField.addActionListener( this );
35         container.add( hourLabel );
36         container.add( hourField );
37
38         // set up minuteLabel and minuteField
39         minuteLabel = new JLabel( "Set minute" );
40         minuteField = new JTextField( 10 );
41         minuteField.addActionListener( this );
42         container.add( minuteLabel );
43         container.add( minuteField );
44
45         // set up secondLabel and secondField
46         secondLabel = new JLabel( "Set Second" );
47         secondField = new JTextField( 10 );
48         secondField.addActionListener( this );
49         container.add( secondLabel );
50         container.add( secondField );
51
52         // set up displayField
53         displayField = new JTextField( 30 );
```

Fig. 8.9 Using class **Time3**'s *set* and *get* methods (part 1 of 4).

```

54     displayField.setEditable( false );
55     container.add( displayField );
56
57     // set up tickButton
58     tickButton = new JButton( "Add 1 to Second" );
59     tickButton.addActionListener( this );
60     container.add( tickButton );
61
62     updateDisplay(); // update text in displayField
63 }
64
65 // handle button and text field events
66 public void actionPerformed( ActionEvent actionEvent )
67 {
68     // process tickButton event
69     if ( actionEvent.getSource() == tickButton )
70         tick();
71
72     // process hourField event
73     else if ( actionEvent.getSource() == hourField ) {
74         time.setHour(
75             Integer.parseInt( actionEvent.getActionCommand() ) );
76         hourField.setText( "" );
77     }
78
79     // process minuteField event
80     else if ( actionEvent.getSource() == minuteField ) {
81         time.setMinute(
82             Integer.parseInt( actionEvent.getActionCommand() ) );
83         minuteField.setText( "" );
84     }
85
86     // process secondField event
87     else if ( actionEvent.getSource() == secondField ) {
88         time.setSecond(
89             Integer.parseInt( actionEvent.getActionCommand() ) );
90         secondField.setText( "" );
91     }
92
93     updateDisplay(); // update displayField and status bar
94 }
95
96 // update displayField and applet container's status bar
97 public void updateDisplay()
98 {
99     displayField.setText( "Hour: " + time.getHour() +
100        "; Minute: " + time.getMinute() +
101        "; Second: " + time.getSecond() );
102
103     showStatus( "Standard time is: " + time.toString() +
104        "; Universal time is: " + time.toUniversalString() );
105 }
106

```

Fig. 8.9 Using class **Time3**'s *set* and *get* methods (part 2 of 4).

```

107 // add one to second and update hour/minute if necessary
108 public void tick()
109 {
110     time.setSecond( ( time.getSecond() + 1 ) % 60 );
111
112     if ( time.getSecond() == 0 ) {
113         time.setMinute( ( time.getMinute() + 1 ) % 60 );
114
115         if ( time.getMinute() == 0 )
116             time.setHour( ( time.getHour() + 1 ) % 24 );
117     }
118 }
119
120 } // end class TimeTest5

```

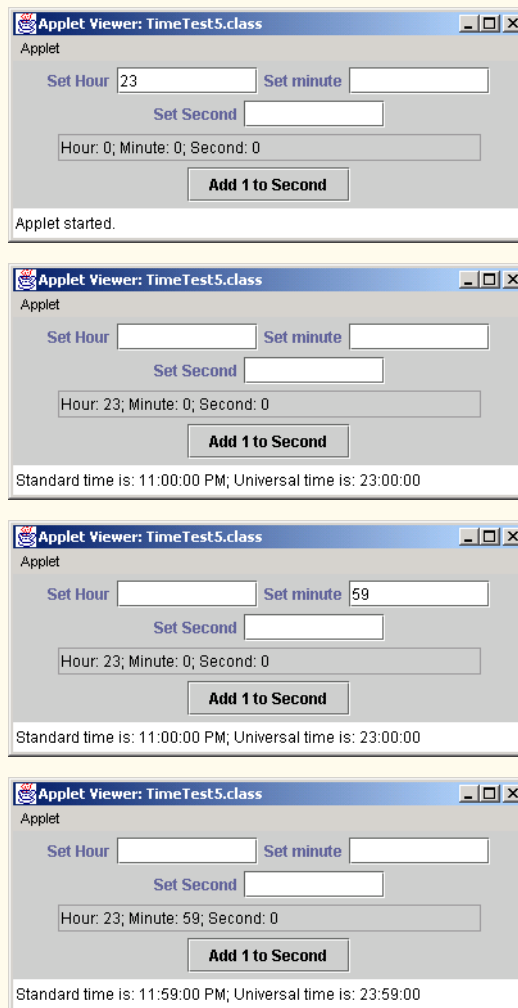


Fig. 8.9 Using class **Time3**'s *set* and *get* methods (part 3 of 4).

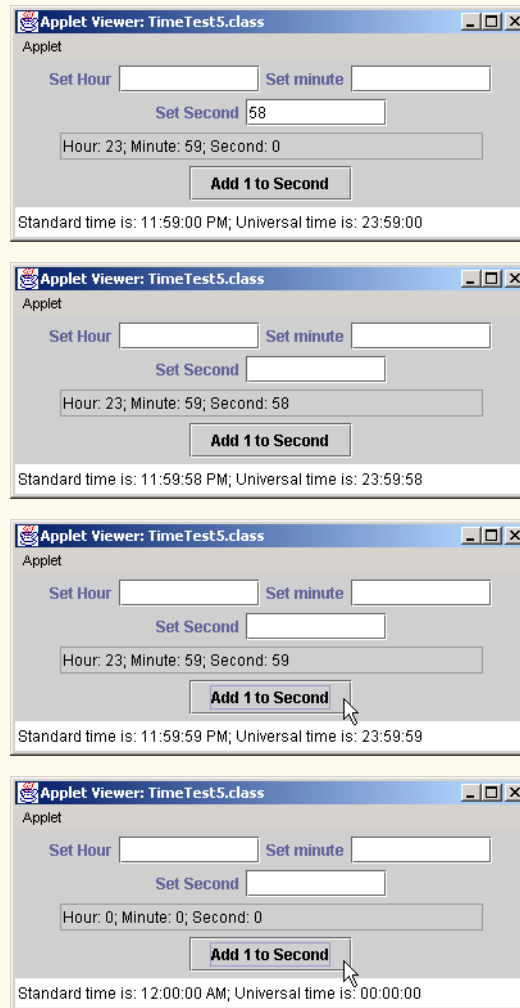


Fig. 8.9 Using class **Time3**'s *set* and *get* methods (part 4 of 4).

Note that when the “**Add 1 to second**” button is clicked, method **actionPerformed** calls the applet's **tick** method (defined at lines 108–118). Method **tick** uses all the new *set* and *get* methods to increment the second properly. Although this works, it incurs the performance burden of issuing multiple method calls. In Section 8.12, we discuss the notion of package access as a means of eliminating this performance burden.



Common Programming Error 8.8

*A constructor can call other methods of the class, such as *set* or *get* methods. However, the instance variables might not yet be in a consistent state, because the constructor is in the process of initializing the object. Using instance variables before they have been initialized properly is a logic error.*

Set methods are certainly important from a software engineering standpoint, because they can perform validity checking. *Set* and *get* methods have another important software engineering advantage, discussed in the following *Software Engineering Observation*.



Software Engineering Observation 8.16

Accessing **private** data through *set* and *get* methods not only protects the instance variables from receiving invalid values, but also insulates clients of the class from the representation of the instance variables. Thus, if the representation of the data changes (typically, to reduce the amount of storage required, improve performance or enhance the class in other ways), only the method implementations need to change—the clients need not change as long as the interface provided by the methods remains the same.

8.8.1 Executing an Applet that Uses Programmer-Defined Packages

After compiling the classes in Fig. 8.8 and Fig. 8.9, you can execute the applet from a command window with the command

```
appletviewer TimeTest5.html
```

As we discussed when we introduced packages earlier in this chapter, the interpreter can locate packaged classes in the current directory. The **appletviewer** is a Java application that executes a Java applet. Like the interpreter, the **appletviewer** can load standard Java classes and extension classes installed on the local computer. However, the **appletviewer** does not use the class path to locate classes in programmer-defined packages. For an applet, such classes should be bundled with the applet class in an archive file called a *Java Archive (JAR)* file. Remember that applets normally are downloaded from the Internet into a Web browser (see Chapter 3 for more information). Bundling the classes and packages that compose an applet enables the applet and its supporting classes to be downloaded as a unit, then executed in the browser (or via the Java Plug-in for browsers that do not support Java 2).

To bundle the classes in Fig. 8.8 and Fig. 8.9, open a command window and change directories to the location in which **TimeTest5.class** is stored. In that same directory should be the **com** directory that begins the package directory structure for class **Time3**. In that directory, issue the following command

```
jar cf TimeTest5.jar TimeTest5.class com\*.*
```

to create the JAR file. [Note: This command uses \ as the directory separator from the MS-DOS prompt. UNIX would use / as the directory separator.] In the preceding command, **jar** is the *Java archive utility* used to create JAR files. Next are the options for the **jar** utility—**cf**. The letter **c** indicates that we are creating a JAR file. The letter **f** indicates that the next argument in the command line (**TimeTest5.jar**) is the name of the JAR file to create. Following the options and JAR file name are the actual files that will be included in the JAR file. We specified **TimeTest5.class** and **com*.***, indicating that **TimeTest5.class** and all the files in the **com** directory should be included in the JAR file. The **com** directory begins the package that contains the **.class** file for the **Time3**. [Note: You can include selected files by specifying the path and file name for each individ-

ual file.] It is important that the directory structure in the JAR file match the directory structure for the packaged classes. Therefore, we executed the **jar** command from the directory in which **com** is located.

To confirm that the files were archived directly, you can issue the command

```
jar tvf TimeTest5.jar
```

which produces the listing in Fig. 8.10. In the preceding command, the options for the **jar** utility are **tvf**. The letter **t** indicates that the table of contents for the JAR should be listed. The letter **v** indicates that the output should be verbose (the verbose output includes the file size in bytes and the date and time each file was created, in addition to the directory structure and file name). The letter **f** specifies that the next argument on the command line is the JAR file to use.

The only remaining issue is to specify the archive as part of the applet's HTML file. In prior examples, **<applet>** tags had the form

```
<applet code = "ClassName.class" width = "width" height = "height">
</applet>
```

To specify that the applet classes are located in a JAR file, use an **<applet>** tag of the form:

```
<applet code = "ClassName.class" archive = "archiveList"
width = "width" height = "height">
</applet>
```

The **archive** attribute can specify a comma-separated list of archive files for use in an applet. Each file in the **archive** list will be downloaded by the browser when it encounters the **<applet>** tags in the HTML document. For the TimeTest5 applet, the applet tag would be

```
<applet code = "TimeTest5.class" archive = "TimeTest5.jar"
width = "400" height = "115">
</applet>
```

Try loading this applet into your Web browser. Remember that you *must* either have a browser that supports Java 2 (such as Netscape Navigator 6) or convert the HTML file for use with the Java Plug-in (as discussed in Chapter 3).

```
0 Fri May 25 14:13:14 EDT 2001 META-INF/
71 Fri May 25 14:13:14 EDT 2001 META-INF/MANIFEST.MF
2959 Fri May 25 13:42:32 EDT 2001 TimeTest5.class
0 Fri May 18 17:35:18 EDT 2001 com/deitel/
0 Fri May 18 17:35:18 EDT 2001 com/deitel/jhttp4/
0 Fri May 18 17:35:18 EDT 2001 com/deitel/jhttp4/ch08/
1765 Fri May 18 17:35:18 EDT 2001 com/deitel/jhttp4/ch08/Time3.class
```

Fig. 8.10 Contents of **TimeTest5.jar**.

8.9 Software Reusability

Java programmers concentrate on crafting new classes and reusing existing classes. Many *class libraries* exist, and others are being developed worldwide. Software is then constructed from existing, well-defined, carefully tested, well-documented, portable, widely available components. This kind of software reusability speeds the development of powerful, high-quality software. *Rapid applications development (RAD)* is of great interest today.

Java programmers now have thousands of classes in the Java API from which to choose to help them implement Java programs. Indeed, Java is not just a programming language. It is a framework in which Java developers can work to achieve true reusability and rapid applications development. Java programmers can focus on the task at hand when developing their programs and leave the lower-level details to the classes of the Java API. For example, to write a program that draws graphics, a Java programmer does not require knowledge of graphics on every computer platform where the program will execute. Instead, a Java programmer concentrates on learning Java's graphics capabilities (which are quite substantial and growing) and writes a Java program that draws the graphics, using Java's API classes such as **Graphics**. When the program executes on a given computer, it is the job of the interpreter to translate Java commands into commands that the local computer can understand.

The Java API classes enable Java programmers to bring new applications to market faster by using preexisting, tested components. Not only does this reduce development time, it also improves programmers' ability to debug and maintain applications. To take advantage of Java's many capabilities, it is essential that programmers take the time to familiarize themselves with the variety of packages and classes in the Java API. There are many Web-based resources at java.sun.com to help you with this task. The primary resource for learning about the Java API is the *Java API documentation*, which can be found at

java.sun.com/j2se/1.3/docs/api/index.html

In addition, java.sun.com provides many other resources, including tutorials, articles and sites specific to individual Java topics. Java developers should also register (for free) at the Java Developer Connection

developer.java.sun.com

This site provides additional resources that Java developers will find useful, including more tutorials and articles, and links to other Java resources. See Appendix B for a more complete list of Java-related Internet and World Wide Web resources.



Good Programming Practice 8.6

Avoid reinventing the wheel. Study the capabilities of the Java API. If the API already contains a class that meets the requirements of your program, use that class rather than creating your own.

In general, to realize the full potential of software reusability, we need to improve cataloging schemes, licensing schemes, protection mechanisms that ensure master copies of classes are not corrupted, description schemes that system designers use to determine if existing objects meet their needs, browsing mechanisms that determine what classes are

available and how closely they meet software developer requirements, and the like. Many interesting research and development problems have been solved and many more need to be solved; these problems will be solved because the potential value of software reuse is enormous.

8.10 Final Instance Variables

We have emphasized repeatedly the *principle of least privilege* as one of the most fundamental principles of good software engineering. Let us see one way in which this principle applies to instance variables.

Some instance variables need to be modifiable and some do not. The programmer can use the keyword **final** to specify that a variable is not modifiable and that any attempt to modify the variable is an error. For example,

```
private final int INCREMENT = 5;
```

declares a constant instance variable **INCREMENT** of type **int** and initializes it to 5.



Software Engineering Observation 8.17

Declaring an instance variable as **final** helps enforce the principle of least privilege. If an instance variable should not be modified, declare it to be **final** to expressly forbid modification.



Testing and Debugging Tip 8.3

Accidental attempts to modify a **final** instance variable are caught at compile time rather than causing execution-time errors. It is always preferable to get bugs out at compile time, if possible, rather than allowing them to slip through to execution time (where studies have found that the cost of repair is often as much as ten times more expensive).



Common Programming Error 8.9

Attempting to modify a **final** instance variable after it is initialized is a syntax error.

The applet of Fig. 8.11 creates a **final** instance variable **INCREMENT** of type **int** and initializes it to 5 in its declaration (line 15). A **final** variable cannot be modified by assignment after it is initialized. Such a variable can be initialized in its declaration or in every constructor of the class.



Common Programming Error 8.10

Not initializing a **final** instance variable in its declaration or in every constructor of the class is a syntax error.

```
1 // Fig. 8.11: Increment.java
2 // Initializing a final variable
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7
```

Fig. 8.11 Initializing a **final** variable (part 1 of 2).

```

8  // Java extension packages
9  import javax.swing.*;
10
11 public class Increment extends JApplet
12     implements ActionListener {
13
14     private int count = 0, total = 0;
15     private final int INCREMENT = 5;    // constant variable
16
17     private JButton button;
18
19     // set up GUI
20     public void init()
21     {
22         Container container = getContentPane();
23
24         button = new JButton( "Click to increment" );
25         button.addActionListener( this );
26         container.add( button );
27     }
28
29     // add INCREMENT to total when user clicks button
30     public void actionPerformed((ActionEvent actionEvent) )
31     {
32         total += INCREMENT;
33         count++;
34         showStatus( "After increment " + count +
35                   ": total = " + total );
36     }
37
38 } // end class Increment

```

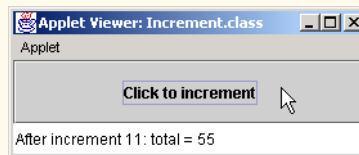


Fig. 8.11 Initializing a **final** variable (part 2 of 2).

Figure 8.12 illustrates compiler errors produced for the program of Fig. 8.11 if instance variable **INCREMENT** is declared **final**, but is not initialized in the declaration.

```

Increment.java:11: variable INCREMENT might not have been
    initialized
public class Increment extends JApplet
    ^
1 error

```

Fig. 8.12 Compiler error message as a result of not initializing increment.

8.11 Composition: Objects as Instance Variables of Other Classes

An **AlarmClock** class object needs to know when it is supposed to sound its alarm, so why not include a reference to a **Time** object as a member of the **AlarmClock** object? Such a capability is called *composition*. A class can have references to objects of other classes as members.



Software Engineering Observation 8.18

One form of software reuse is composition, in which a class has references to objects of other classes as members.

The next program contains three classes—**Date** (Fig. 8.13), **Employee** (Fig. 8.14) and **EmployeeTest** (Fig. 8.15). Class **Employee** contains instance variables **firstName**, **lastName**, **birthDate** and **hireDate**. Members **birthDate** and **hireDate** are references to **Dates** that contain instance variables **month**, **day** and **year**. This demonstrates that a class can contain references to objects of other classes. Class **EmployeeTest** instantiates an **Employee** and initializes and displays its instance variables. The **Employee** constructor (Fig. 8.14, lines 12–20) takes eight arguments—**first**, **last**, **birthMonth**, **birthDay**, **birthYear**, **hireMonth**, **hireDay** and **hireYear**. Arguments **birthMonth**, **birthDay** and **birthYear** are passed to the **Date** constructor (Fig. 8.13, lines 13–28) to initialize the **birthDate** object and **hireMonth**, **hireDay** and **hireYear** are passed to the **Date** constructor to initialize the **hireDate** object.

A member object does not need to be initialized immediately with constructor arguments. If an empty argument list is provided when a member object is created, the object's default constructor (or no-argument constructor if one is available) will be called automatically. Values, if any, established by the default constructor (or no-argument constructor) can then be replaced by *set* methods.



Performance Tip 8.2

Initialize member objects explicitly at construction time by passing appropriate arguments to the constructors of the member objects. This eliminates the overhead of initializing member objects twice—once when the member object's default constructor is called and again when set methods are used to provide initial values for the member object.

Note that both class **Date** (Fig. 8.13) and class **Employee** (Fig. 8.14) are defined as part of the package **com.deitel.jhtp4.ch08** as specified on line 3 of each file. Because they are in the same package (i.e., the same directory), class **Employee** does not need to import class **Date** to use it. When the compiler searches for the file **Date.class**, the compiler knows to search the directory where **Employee.class** is located. Classes in a package never need to import other classes from the same package.

```

1  // Fig. 8.13: Date.java
2  // Declaration of the Date class.
3  package com.deitel.jhtp4.ch08;
4
5  public class Date extends Object {
6      private int month; // 1-12

```

Fig. 8.13 **Date** class (part 1 of 2).

```

7  private int day;      // 1-31 based on month
8  private int year;     // any year
9
10 // Constructor: Confirm proper value for month;
11 // call method checkDay to confirm proper
12 // value for day.
13 public Date( int theMonth, int theDay, int theYear )
14 {
15     if ( theMonth > 0 && theMonth <= 12 ) // validate month
16         month = theMonth;
17     else {
18         month = 1;
19         System.out.println( "Month " + theMonth +
20                             " invalid. Set to month 1." );
21     }
22
23     year = theYear;           // could validate year
24     day = checkDay( theDay ); // validate day
25
26     System.out.println(
27         "Date object constructor for date " + toString() );
28 }
29
30 // Utility method to confirm proper day value
31 // based on month and year.
32 private int checkDay( int testDay )
33 {
34     int daysPerMonth[] =
35         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
36
37     // check if day in range for month
38     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39         return testDay;
40
41     // check for leap year
42     if ( month == 2 && testDay == 29 &&
43         ( year % 400 == 0 ||
44           ( year % 4 == 0 && year % 100 != 0 ) ) )
45         return testDay;
46
47     System.out.println( "Day " + testDay +
48                         " invalid. Set to day 1." );
49
50     return 1; // leave object in consistent state
51 }
52
53 // Create a String of the form month/day/year
54 public String toString()
55 {
56     return month + "/" + day + "/" + year;
57 }
58
59 } // end class Date

```

Fig. 8.13 Date class (part 2 of 2).

```
1 // Fig. 8.14: Employee.java
2 // Definition of class Employee.
3 package com.deitel.jhtp4.ch08;
4
5 public class Employee extends Object {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11     // constructor to initialize name, birth date and hire date
12     public Employee( String first, String last,
13         int birthMonth, int birthDay, int birthYear,
14         int hireMonth, int hireDay, int hireYear )
15     {
16         firstName = first;
17         lastName = last;
18         birthDate = new Date( birthMonth, birthDay, birthYear );
19         hireDate = new Date( hireMonth, hireDay, hireYear );
20     }
21
22     // convert Employee to String format
23     public String toString()
24     {
25         return lastName + ", " + firstName +
26             "   Hired: " + hireDate.toString() +
27             "   Birthday: " + birthDate.toString();
28     }
29
30 } // end class Employee
```

Fig. 8.14 **Employee** class with member object references.

```
1 // Fig. 8.15: EmployeeTest.java
2 // Demonstrating an object with a member object.
3
4 // Java extension packages
5 import javax.swing.JOptionPane;
6
7 // Deitel packages
8 import com.deitel.jhtp4.ch08.Employee;
9
10 public class EmployeeTest {
11
12     // test class Employee
13     public static void main( String args[] )
14     {
15         Employee employee = new Employee( "Bob", "Jones",
16             7, 24, 49, 3, 12, 88 );
17     }
18 }
```

Fig. 8.15 Demonstrating an object with a member object reference (part 1 of 2).

```

18     JOptionPane.showMessageDialog( null,
19         employee.toString(), "Testing Class Employee",
20         JOptionPane.INFORMATION_MESSAGE );
21
22     System.exit( 0 );
23 }
24
25 } // end class EmployeeTest

```



Date object constructor for date 7/24/1949
 Date object constructor for date 3/12/1988

Fig. 8.15 Demonstrating an object with a member object reference (part 2 of 2).

8.12 Package Access

When no member access modifier is provided for a method or variable when it is defined in a class, the method or variable is considered to have *package access*. If your program consists of one class definition, this has no specific effects on the program. However, if your program uses multiple classes from the same package (i.e., a group of related classes), these classes can access each other's package-access methods and data directly through a reference to an object.



Performance Tip 8.3

Package access enables objects of different classes to interact without the need for set and get methods that provide access to data, thus eliminating some of the method call overhead.

Let us consider a mechanical example of package access. The application of Fig. 8.16 contains two classes—the **PackageDataTest** application class (lines 8–34) and the **PackageData** class (lines 37–54). In the **PackageData** class definition, lines 38–39 declare the instance variables **number** and **string** with no member access modifiers; therefore, these are package access instance variables. The **PackageDataTest** application's **main** method creates an instance of the **PackageData** class (line 13) to demonstrate the ability to modify the **PackageData** instance variables directly (as shown on lines 20–21). The results of the modification can be seen in the output window.

When you compile this program, the compiler produces two separate files—a **.class** file for class **PackageData** and a **.class** file for class **PackageDataTest**. Every Java class has its own **.class** file. These two **.class** files are placed in the same directory by the compiler automatically and are considered to be part of the same package (they are certainly related by the fact that they are in the same file). Because they are part of the same package, class **PackageDataTest** is allowed to modify the package access data of objects of class **PackageData**.

```
1 // Fig. 8.16: PackageDataTest.java
2 // Classes in the same package (i.e., the same directory) can
3 // use package access data of other classes in the same package.
4
5 // Java extension packages
6 import javax.swing.JOptionPane;
7
8 public class PackageDataTest {
9
10     // Java extension packages
11     public static void main( String args[] )
12     {
13         PackageData packageData = new PackageData();
14
15         // append String representation of packageData to output
16         String output =
17             "After instantiation:\n" + packageData.toString();
18
19         // change package access data in packageData object
20         packageData.number = 77;
21         packageData.string = "Good bye";
22
23         // append String representation of packageData to output
24         output += "\nAfter changing values:\n" +
25             packageData.toString();
26
27         JOptionPane.showMessageDialog( null, output,
28             "Demonstrating Package Access",
29             JOptionPane.INFORMATION_MESSAGE );
30
31         System.exit( 0 );
32     }
33 } // end class PackageDataTest
34
35 // class with package access instance variables
36 class PackageData {
37     int number; // package access instance variable
38     String string; // package access instance variable
39
40     // constructor
41     public PackageData()
42     {
43         number = 0;
44         string = "Hello";
45     }
46
47     // convert PackageData object to String representation
48     public String toString()
49     {
50         return "number: " + number + "    string: " + string;
51     }
52 }
53
```

Fig. 8.16 Package access to members of a class (part 1 of 2).

```
54 } // end class PackageData
```

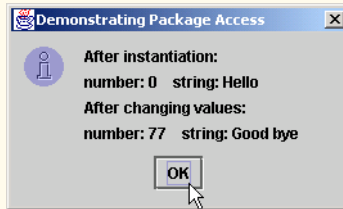


Fig. 8.16 Package access to members of a class (part 2 of 2).



Software Engineering Observation 8.19

Some people in the OOP community feel that package access corrupts information hiding and weakens the value of the object-oriented design approach, because the programmer must assume responsibility for error checking and data validation in any code that manipulates the package access data members.

8.13 Using the **this** Reference

When a method of a class references another member of that class for a specific object of that class, how does Java ensure that the proper object is referenced? The answer is that each object has access to a reference to itself—called the **this** reference.

The **this** reference is used implicitly to refer to both the instance variables and methods of an object. We begin with a simple example of using the **this** reference explicitly; a subsequent example shows some substantial and subtle examples of using **this**.



Performance Tip 8.4

Java conserves storage by maintaining only one copy of each method per class; this method is invoked by every object of that class. Each object, on the other hand, has its own copy of the class's instance variables.

The application of Fig. 8.17 demonstrates implicit and explicit use of the **this** reference to enable the **main** method of class **ThisTest** to display the **private** data of a **SimpleTime** object.

```
1 // Fig. 8.17: ThisTest.java
2 // Using the this reference to refer to
3 // instance variables and methods.
4
5 // Java core packages
6 import java.text.DecimalFormat;
7
8 // Java extension packages
9 import javax.swing.*;
10
```

Fig. 8.17 Using the **this** reference implicitly and explicitly (part 1 of 3).

```

11 public class ThisTest {
12
13     // test class SimpleTime
14     public static void main( String args[] )
15     {
16         SimpleTime time = new SimpleTime( 12, 30, 19 );
17
18         JOptionPane.showMessageDialog( null, time.buildString(),
19             "Demonstrating the \"this\" Reference",
20             JOptionPane.INFORMATION_MESSAGE );
21
22         System.exit( 0 );
23     }
24 } // end class ThisTest
25
26 // class SimpleTime demonstrates the "this" reference
27 class SimpleTime {
28     private int hour, minute, second;
29
30     // constructor uses parameter names identical to instance
31     // variable names, so "this" reference required to distinguish
32     // between instance variables and parameters
33     public SimpleTime( int hour, int minute, int second )
34     {
35         this.hour = hour;        // set "this" object's hour
36         this.minute = minute;    // set "this" object's minute
37         this.second = second;    // set "this" object's second
38     }
39
40     // call toString explicitly via "this" reference, explicitly
41     // via implicit "this" reference, implicitly via "this"
42     public String buildString()
43     {
44         return "this.toString(): " + this.toString() +
45             "\ntoString(): " + toString() +
46             "\nthis (with implicit toString() call): " + this;
47     }
48
49     // convert SimpleTime to String format
50     public String toString()
51     {
52         DecimalFormat twoDigits = new DecimalFormat( "00" );
53
54         // "this" not required, because toString does not have
55         // local variables with same names as instance variables
56         return twoDigits.format( this.hour ) + ":" +
57             twoDigits.format( this.minute ) + ":" +
58             twoDigits.format( this.second );
59     }
60 }
61
62 } // end class SimpleTime

```

Fig. 8.17 Using the **this** reference implicitly and explicitly (part 2 of 3).

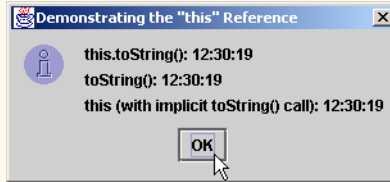


Fig. 8.17 Using the **this** reference implicitly and explicitly (part 3 of 3).

Class **SimpleTime** (lines 8–60) defines three **private** instance variables—**hour**, **minute** and **second**. The constructor (lines 34–39) receives three **int** arguments to initialize a **SimpleTime** object. Notice that the parameter names for the constructor are the same as the instance variable names. Remember that a local variable of a method with the same name as an instance variable of a class hides the instance variable in the scope of the method. For this reason, we use the **this** reference to refer explicitly to the instance variables on lines 36–38.



Common Programming Error 8.11

*In a method in which a method parameter has the same name as one of the class members, use **this** explicitly if you want to access the class member; otherwise, you will incorrectly reference the method parameter.*



Good Programming Practice 8.7

Avoid using method parameter names that conflict with class member names.

Method **buildString** (lines 43–48) returns a **String** created with a statement that uses the **this** reference three ways. Line 45 explicitly invokes the class's **toString** method via **this.toString()**. Line 46 implicitly uses the **this** reference to perform the same task. Line 47 appends **this** to the string that will be returned. Remember that the **this** reference is a reference to an object—the current **SimpleTime** object being manipulated. As before, any reference added to a **String** results in a call to the **toString** method for the referenced object. Method **buildString** is invoked at line 18 to display the results of the three calls to **toString**. Note that the same time is displayed on all three lines of the output because all three calls to **toString** are for the same object.

Another use of the **this** reference is in enabling *concatenated method calls* (also called *cascaded method calls* or *method call chaining*). Figure 8.18 illustrates returning a reference to a **Time4** object to enable method calls of class **Time4** to be concatenated. Methods **setTime** (lines 50–57), **setHour** (lines 60–65), **setMinute** (line 68–74) and **setSecond** (lines 77–83) each have a return type of **Time4** and each has as its last statement

```
return this;
```

to indicate that a reference to the **Time4** object being manipulated should be returned to the caller of the method.

```
1  // Fig. 8.18: Time4.java
2  // Time4 class definition
3  package com.deitel.jhtp4.ch08;
4
5  // Java core packages
6  import java.text.DecimalFormat;
7
8  public class Time4 extends Object {
9      private int hour;        // 0 - 23
10     private int minute;      // 0 - 59
11     private int second;      // 0 - 59
12
13     // Time4 constructor initializes each instance variable
14     // to zero. Ensures that Time object starts in a
15     // consistent state.
16     public Time4()
17     {
18         this.setTime( 0, 0, 0 );
19     }
20
21     // Time4 constructor: hour supplied, minute and second
22     // defaulted to 0
23     public Time4( int hour )
24     {
25         this.setTime( hour, 0, 0 );
26     }
27
28     // Time4 constructor: hour and minute supplied, second
29     // defaulted to 0
30     public Time4( int hour, int minute )
31     {
32         this.setTime( hour, minute, 0 );
33     }
34
35     // Time4 constructor: hour, minute and second supplied
36     public Time4( int hour, int minute, int second )
37     {
38         this.setTime( hour, minute, second );
39     }
40
41     // Time4 constructor: another Time4 object supplied.
42     public Time4( Time4 time )
43     {
44         this.setTime( time.getHour(), time.getMinute(),
45                     time.getSecond() );
46     }
47
48     // Set Methods
49     // set a new Time value using universal time
50     public Time4 setTime( int hour, int minute, int second )
51     {
52         this.setHour( hour );        // set hour
```

Fig. 8.18 Class **Time4** using **this** to enable chained method calls (part 1 of 3).

```
53         this.setMinute( minute ); // set minute
54         this.setSecond( second ); // set second
55
56         return this; // enables chaining
57     }
58
59     // validate and set hour
60     public Time4 setHour( int hour )
61     {
62         this.hour = ( hour >= 0 && hour < 24 ? hour : 0 );
63
64         return this; // enables chaining
65     }
66
67     // validate and set minute
68     public Time4 setMinute( int minute )
69     {
70         this.minute =
71             ( minute >= 0 && minute < 60 ) ? minute : 0;
72
73         return this; // enables chaining
74     }
75
76     // validate and set second
77     public Time4 setSecond( int second )
78     {
79         this.second =
80             ( second >= 0 && second < 60 ) ? second : 0;
81
82         return this; // enables chaining
83     }
84
85     // Get Methods
86     // get value of hour
87     public int getHour() { return this.hour; }
88
89     // get value of minute
90     public int getMinute() { return this.minute; }
91
92     // get value of second
93     public int getSecond() { return this.second; }
94
95     // convert to String in universal-time format
96     public String toUniversalString()
97     {
98         DecimalFormat twoDigits = new DecimalFormat( "00" );
99
100         return twoDigits.format( this.getHour() ) + ":" +
101             twoDigits.format( this.getMinute() ) + ":" +
102             twoDigits.format( this.getSecond() );
103     }
104
```

Fig. 8.18 Class **Time4** using **this** to enable chained method calls (part 2 of 3).

```

105 // convert to String in standard-time format
106 public String toString()
107 {
108     DecimalFormat twoDigits = new DecimalFormat( "00" );
109
110     return ( this.getHour() == 12 || this.getHour() == 0 ?
111         12 : this.getHour() % 12 ) + ":" +
112         twoDigits.format( this.getMinute() ) + ":" +
113         twoDigits.format( this.getSecond() ) +
114         ( this.getHour() < 12 ? " AM" : " PM" );
115 }
116
117 } // end class Time4

```

Fig. 8.18 Class **Time4** using **this** to enable chained method calls (part 3 of 3).

The example again demonstrates the explicit use of the **this** reference inside the body of a class. In class **Time4**, every use of an instance variable of the class and every call to another method in class **Time4** uses the **this** reference explicitly. Most programmers prefer not to use the **this** reference unless it is required or helps clarify a piece of code.



Good Programming Practice 8.8

Explicitly using **this** can increase program clarity in some contexts in which **this** is optional.

In application class **TimeTest6** (Fig. 8.19), line 18 and line 26 both demonstrate method call chaining. Why does the technique of returning the **this** reference work? Let us discuss line 18. The dot operator (.) associates from left to right, so the expression

```
time.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
```

first evaluates **time.setHour(18)**, then returns a reference to object **time** as the result of this method call. Any time you have a reference in a program (even as the result of a method call), the reference can be followed by a dot operator and a call to one of the methods of the reference type. Thus, the remaining expression is interpreted as

```
time.setMinute( 30 ).setSecond( 22 );
```

The **time.setMinute(30)** call executes and returns a reference to **time**. The remaining expression is interpreted as

```
time.setSecond( 22 );
```

When the statement is complete, the time is **18** for the **hour**, **30** for the **minute** and **22** from the **second**.

Note that the calls on line 26

```
time.setTime( 20, 20, 20 ).toString();
```

also use the concatenation feature. These method calls must appear in this order in this expression because **toString** as defined in the class does not return a reference to a **Time4** object. Placing the call to **toString** before the call to **setTime** causes a syntax error. Note that **toString** returns a reference to a **String** object. Therefore, a method of class **String** could be concatenated to the end of line 26.

```

1  // Fig. 8.19: TimeTest6.java
2  // Chaining method calls together with the this reference
3
4  // Java extension packages
5  import javax.swing.*;
6
7  // Deitel packages
8  import com.deitel.jhtp4.ch08.Time4;
9
10 public class TimeTest6 {
11
12     // test method call chaining with object of class Time4
13     public static void main( String args[] )
14     {
15         Time4 time = new Time4();
16
17         // chain calls to setHour, setMinute and setSecond
18         time.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
19
20         // use method call chaining to set new time and get
21         // String representation of new time
22         String output =
23             "Universal time: " + time.toUniversalString() +
24             "\nStandard time: " + time.toString() +
25             "\n\nNew standard time: " +
26             time.setTime( 20, 20, 20 ).toString();
27
28         JOptionPane.showMessageDialog( null, output,
29             "Chaining Method Calls",
30             JOptionPane.INFORMATION_MESSAGE );
31
32         System.exit( 0 );
33     }
34
35 } // end class TimeTest6

```

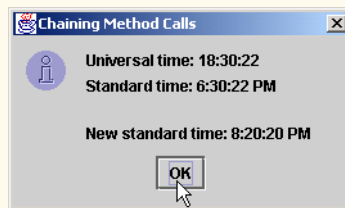


Fig. 8.19 Concatenating method calls.

Note that the purpose of the example in Fig. 8.18 and Fig. 8.19 is to demonstrate the mechanics of concatenated method calls. Many Java methods return references to objects that can be used in the manner shown here. It is important to understand concatenated method calls as they appear frequently in Java programs.



Good Programming Practice 8.9

For program clarity, avoid using concatenated method calls.

8.14 Finalizers

We have seen that constructor methods are capable of initializing data in an object of a class when the class is created. Often, constructors acquire various system resources such as memory (when the **new** operator is used). We need a disciplined way to give resources back to the system when they are no longer needed to avoid resource leaks. The most common resource acquired by constructors is memory. Java performs automatic *garbage collection* of memory to help return memory back to the system. When an object is no longer used in the program (i.e., there are no references to the object), the object is *marked for garbage collection*. The memory for such an object can be reclaimed when the *garbage collector* executes. Therefore, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely to happen in Java. However, other resource leaks can occur.



Performance Tip 8.5

Extensive use of local variables that refer to objects can degrade performance. When a local variable is the only reference to an object, the object is marked for garbage collection as the local variable goes out of scope. If this happens frequently in short time periods, large numbers of objects could be marked for garbage collection, thus placing a performance burden on the garbage collector.

Every class in Java can have a *finalizer method* that returns resources to the system. The finalizer method for an object is guaranteed to be called to perform *termination house-keeping* on the object just before the garbage collector reclaims the memory for the object. A class's finalizer method always has the name **finalize**, receives no parameters and returns no value (i.e., its return type is **void**). A class should have only one **finalize** method that takes no arguments. Method **finalize** is defined originally in class **Object** as a placeholder that does nothing. This guarantees that every class has a **finalize** method for the garbage collector to call.



Good Programming Practice 8.10

*The last statement in a **finalize** method should always be **super.finalize()**; to ensure that the superclass's **finalize** method is called.*



Software Engineering Observation 8.20

*The garbage collector is not guaranteed to execute; therefore, an object's **finalize** method is not guaranteed to get called. You should not architect classes that rely on the garbage collector calling an object's **finalize** method to deallocate resources.*

Finalizers have not been provided for the classes presented so far. Actually, finalizers are rarely used in industrial Java applications. We will see a sample **finalize** method and discuss the garbage collector further in Figure 8.20.



Testing and Debugging Tip 8.4

*Several professional Java developers who reviewed this book indicated that method **finalize** is not useful in industrial Java applications, because it is not guaranteed to get called. For this reason, you should search your Java programs for any use of method **finalize** to ensure that the program does not rely on calls to method **finalize** for proper deallocation of resources. In fact, you might consider removing method **finalize** entirely and*

using other techniques to ensure proper resource deallocation. We present one such technique in Chapter 14, Exception Handling.

8.15 Static Class Members

Each object of a class has its own copy of all the instance variables of the class. In certain cases, only one copy of a particular variable should be shared by all objects of a class. A **static class variable** is used for these and other reasons. A **static** class variable represents *class-wide information*—all objects of the class share the same piece of data. The declaration of a **static** member begins with the keyword **static**.

Let us motivate the need for **static** class-wide data with a video game example. Suppose we have a video game with **Martians** and other space creatures. Each **Martian** tends to be brave and willing to attack other space creatures when the **Martian** is aware that there are at least five **Martians** present. If there are fewer than five **Martians** present, each **Martian** becomes cowardly. So each **Martian** needs to know the **martianCount**. We could endow class **Martian** with **martianCount** as instance data. If we do this, then every **Martian** will have a separate copy of the instance data and every time we create a new **Martian** we will have to update the instance variable **martianCount** in every **Martian**. This wastes space with the redundant copies and wastes time in updating the separate copies. Instead, we declare **martianCount** to be **static**. This makes **martianCount** class-wide data. Every **Martian** can see the **martianCount** as if it were instance data of the **Martian**, but only one copy of the static **martianCount** is maintained by Java. This saves space. We save time by having the **Martian** constructor increment the static **martianCount**. Because there is only one copy, we do not have to increment separate copies of **martianCount** for each **Martian** object.



Performance Tip 8.6

Use **static** class variables to save storage when a single copy of the data will suffice.

Although **static** class variables may seem like global variables, **static** class variables have class scope. A class's **public static** class members can be accessed through a reference to any object of that class, or they can be accessed through the class name by using the dot operator (e.g., **Math.random()**). A class's **private static** class members can be accessed only through methods of the class. Actually, **static** class members exist even when no objects of that class exist—they are available as soon as the class is loaded into memory at execution time. To access a **public static** class member when no objects of the class exist, simply prefix the class name and the dot operator to the class member. To access a **private static** class member when no objects of the class exist, a **public static** method must be provided and the method must be called by prefixing its name with the class name and dot operator.

Our next program defines two classes—**Employee** (Fig. 8.20) and **EmployeeTest** (Fig. 8.21). Class **Employee** defines a **private static** class variable and a **public static** method. The class variable **count** (Fig. 8.20, line 6) is initialized to zero by default. Class variable **count** maintains a count of the number of objects of class **Employee** that have been instantiated and currently reside in memory. This includes objects that have already been marked for garbage collection but have not yet been reclaimed.

```

1  // Fig. 8.20: Employee.java
2  // Employee class definition.
3  public class Employee extends Object {
4      private String firstName;
5      private String lastName;
6      private static int count; // number of objects in memory
7
8      // initialize employee, add 1 to static count and
9      // output String indicating that constructor was called
10     public Employee( String first, String last )
11     {
12         firstName = first;
13         lastName = last;
14
15         ++count; // increment static count of employees
16         System.out.println( "Employee object constructor: " +
17             firstName + " " + lastName );
18     }
19
20     // subtract 1 from static count when garbage collector
21     // calls finalize to clean up object and output String
22     // indicating that finalize was called
23     protected void finalize()
24     {
25         --count; // decrement static count of employees
26         System.out.println( "Employee object finalizer: " +
27             firstName + " " + lastName + "; count = " + count );
28     }
29
30     // get first name
31     public String getFirstName()
32     {
33         return firstName;
34     }
35
36     // get last name
37     public String getLastName()
38     {
39         return lastName;
40     }
41
42     // static method to get static count value
43     public static int getCount()
44     {
45         return count;
46     }
47
48 } // end class Employee

```

Fig. 8.20 **Employee** class that uses a **static** class variable to maintain a count of the number of **Employee** objects in memory.

When objects of class **Employee** exist, member **count** can be used in any method of an **Employee** object—in this example, **count** is incremented (line 15) by the con-

structor and decremented (line 25) by the finalizer. When no objects of class **Employee** exist, member **count** can still be referenced, but only through a call to **public static** method **getCount** (lines 43–46), as in:

```
Employee.getCount()
```

which determines the number of **Employee** objects currently in memory. Note that when there are no objects instantiated in the program, the **Employee.getCount()** method call is issued. However, when there are objects instantiated, method **getCount** can also be called through a reference to one of the objects, as in

```
e1.getCount()
```



Good Programming Practice 8.11

Always invoke **static** methods by using the class name and the dot operator (.). This emphasizes to other programmers reading your code that the method being called is a **static** method.

Notice that the **Employee** class has a **finalize** method (lines 23–28). This method is included to show when it is called by the garbage collector in a program. Method **finalize** normally is declared **protected** so it is not part of the **public** services of a class. We will discuss the **protected** access modifier in detail in Chapter 9.

Method **main** of the **EmployeeTest** application class (Fig. 8.21) instantiates two **Employee** objects (lines 16–17). When each **Employee** object's constructor is invoked, lines 12–13 of Fig. 8.20 store references to that **Employee**'s first name and last name **String** objects. Note that these two statements *do not* make copies of the original **Strings** arguments. Actually, **String** objects in Java are *immutable*—they cannot be modified after they are created (class **String** does not provide any *set* methods). Because a reference cannot be used to modify a **String**, it is safe to have many references to one **String** object in a Java program. This is not normally the case for most other classes in Java. If Java **String** objects are immutable, why are we able to use the **+** and **+=** operators to concatenate **Strings**? As we discuss in Chapter 10, **Strings** and **Characters**, the **String** concatenation operations actually result in the creation of new **String** objects containing the concatenated values. The original **String** objects actually are not modified.

When **main** is done with the two **Employee** objects, the references **e1** and **e2** are set to **null** at lines 37–38. At this point references **e1** and **e2** no longer refer to the objects that were instantiated on lines 16–17. This *marks the objects for garbage collection* because there are no more references to the objects in the program.

```
1 // Fig. 8.21: EmployeeTest.java
2 // Test Employee class with static class variable,
3 // static class method, and dynamic memory.
4 import javax.swing.*;
5
```

Fig. 8.21 Using a **static** class variable to maintain a count of the number of objects of a class (part 1 of 3).

```

6  public class EmployeeTest {
7
8      // test class Employee
9      public static void main( String args[] )
10     {
11         // prove that count is 0 before creating Employees
12         String output = "Employees before instantiation: " +
13             Employee.getCount();
14
15         // create two Employees; count should be 2
16         Employee e1 = new Employee( "Susan", "Baker" );
17         Employee e2 = new Employee( "Bob", "Jones" );
18
19         // Prove that count is 2 after creating two Employees.
20         // Note: static methods should be called only via the
21         // class name for the class in which they are defined.
22         output += "\n\nEmployees after instantiation: " +
23             "\nvia e1.getCount(): " + e1.getCount() +
24             "\nvia e2.getCount(): " + e2.getCount() +
25             "\nvia Employee.getCount(): " + Employee.getCount();
26
27         // get names of Employees
28         output += "\n\nEmployee 1: " + e1.getFirstName() +
29             " " + e1.getLastName() + "\nEmployee 2: " +
30             e2.getFirstName() + " " + e2.getLastName();
31
32         // If there is only one reference to each employee (as
33         // on this example), the following statements mark
34         // those objects for garbage collection. Otherwise,
35         // these statement simply decrement the reference count
36         // for each object.
37         e1 = null;
38         e2 = null;
39
40         System.gc(); // suggest call to garbage collector
41
42         // Show Employee count after calling garbage collector.
43         // Count displayed may be 0, 1 or 2 depending on
44         // whether garbage collector executed immediately and
45         // number of Employee objects it collects.
46         output += "\n\nEmployees after System.gc(): " +
47             Employee.getCount();
48
49         JOptionPane.showMessageDialog( null, output,
50             "Static Members and Garbage Collection",
51             JOptionPane.INFORMATION_MESSAGE );
52
53         System.exit( 0 );
54     }
55
56 } // end class EmployeeTest

```

Fig. 8.21 Using a **static** class variable to maintain a count of the number of objects of a class (part 2 of 3).

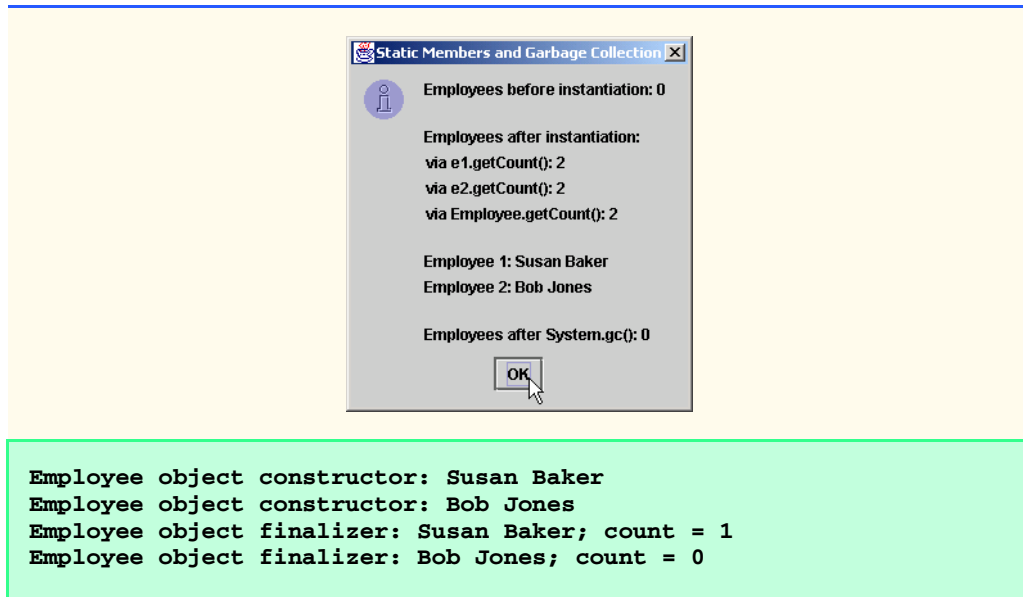


Fig. 8.21 Using a **static** class variable to maintain a count of the number of objects of a class (part 3 of 3).

Eventually, the garbage collector reclaims the memory for these objects (or the memory is reclaimed by the operating system when the program terminates). Because it is not guaranteed when the garbage collector will execute, we make an explicit call to the garbage collector with line 40, which uses **public static** method **gc** from class **System** (**java.lang** package) to indicate that the garbage collector immediately should make a best effort attempt to collect objects that have been marked for garbage collection. However, this is just a best effort—it is possible that no objects or a subset of the garbage objects will be collected. In our example, the garbage collector did execute before lines 49–51 displayed the results of the program. The last line of the output indicates that the number of **Employee** objects in memory is 0 after the call to **System.gc()**. Also, the last two lines of the command window output show that the **Employee** object for **Susan Baker** was finalized before the **Employee** object for **Bob Jones**. Remember, the garbage collector is not guaranteed to execute when **System.gc()** is invoked nor is it guaranteed to collect objects in a specific order, so it is possible that the output of this program on your system may differ.

[Note: A method declared **static** cannot access **nonstatic** class members. Unlike **nonstatic** methods, a **static** method has no **this** reference because, **static** class variables and **static** class methods exist independent of any objects of a class and before any objects of the class have been instantiated.]



Common Programming Error 8.12

Referring to the **this** reference in a **static** method is a syntax error.



Common Programming Error 8.13

It is a syntax error for a **static** method to call an instance method or to access an instance variable.



Software Engineering Observation 8.21

Any **static** class variables and **static** class methods exist and can be used even if no objects of that class have been instantiated.

8.16 Data Abstraction and Encapsulation

Classes normally hide their implementation details from the clients of the classes. This is called *encapsulation* or *information hiding*. As an example of encapsulation, let us consider a data structure called a *stack*.

Think of a stack in terms of a pile of dishes. When a dish is placed on the pile, it is always placed at the top (referred to as *pushing* the dish onto the stack), and when a dish is removed from the pile, it is always removed from the top (referred to as *popping* the dish off the stack). Stacks are known as *last-in, first-out (LIFO) data structures*—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

The programmer can create a stack class and hide from its clients implementation of the stack. Stacks can easily be implemented with arrays and other methods (such as linked lists; see Chapter 19, “Data Structures,” and Chapter 20, “Java Utilities Packages and Bit Manipulation”). A client of a stack class need not know how the stack is implemented. The client simply requires that when data items are placed in the stack, they will be recalled in last-in, first-out order. This concept is referred to as *data abstraction*, and Java classes define abstract data types (ADTs). Although users might happen to know the details of how a class is implemented, users may not write code that depends on these details. This means that a particular class (such as one that implements a stack and its operations of *push* and *pop*) can be replaced with another version without affecting the rest of the system, as long as the public services of that class does not change (i.e., every method still has the same name, return type and parameter list in the new class definition).

The job of a high-level language is to create a view convenient for programmers to use. There is no single accepted standard view—that is one reason why there are so many programming languages. Object-oriented programming in Java presents yet another view.

Most programming languages emphasize actions. In these languages, data exists in support of the actions programs need to take. Data is “less interesting” than actions, anyway. Data is “crude.” There are only a few built-in data types, and it is difficult for programmers to create their own new data types.

This view changes with Java and the object-oriented style of programming. Java elevates the importance of data. The primary activity in Java is creating new data types (i.e., classes) and expressing the interactions among objects of those data types.

To move in this direction, the programming-languages community needed to formalize some notions about data. The formalization we consider is the notion of *abstract data types (ADTs)*. ADTs receive as much attention today as structured programming did over the last two decades. ADTs do not replace structured programming. Rather, they provide an additional formalization to further improve the program development process.

What is an abstract data type? Consider the built-in type **int**. What comes to mind is the notion of an integer in mathematics, but **int** on a computer is not precisely what an integer is in mathematics. In particular, computer **ints** are normally quite limited in size. For example, **int** on a 32-bit machine is limited approximately to the range -2 billion to $+2$ billion. If the result of a calculation falls outside this range, an error occurs and the

machine responds in some machine-dependent manner, including the possibility of “quietly” producing an incorrect result. Mathematical integers do not have this problem. So the notion of a computer **int** is really only an approximation to the notion of a real-world integer. The same is true with **float**.

The point is that even the built-in data types provided with programming languages like Java are really only approximations or models of real-world concepts and behaviors. We have taken **int** for granted until this point, but now you have a new perspective to consider. Types like **int**, **float**, **char** and others are all examples of abstract data types. They are essentially ways of representing real-world notions to some satisfactory level of precision within a computer system.

An abstract data type actually captures two notions, namely a *data representation* and the *operations* that are allowed on that data. For example, the notion of **int** defines addition, subtraction, multiplication, division and modulus operations in Java, but division by zero is undefined. Another example is the notion of negative integers whose operations and data representation are clear, but the operation of taking the square root of a negative integer is undefined. In Java, the programmer uses classes to implement abstract data types.

Java has a small set of primitive types. ADTs extend the base programming language.



Software Engineering Observation 8.22

The programmer is able to create new types through the use of the class mechanism. These new types can be designed to be used as conveniently as the built-in types. Thus, Java is an extensible language. Although the language is easy to extend with these new types, the base language itself is not changeable.

New Java classes can be proprietary to an individual, to small groups, to companies, and so on. Many classes are placed in standard *class libraries* intended for wide distribution. This does not necessarily promote standards, although de facto standards are emerging. The full value of Java will be realized only when substantial, standardized class libraries become more widely available than they are today. In the United States, such standardization often happens through *ANSI*, the *American National Standards Institute*. Worldwide standardization often happens through *ISO*, the *International Standards Organization*. Regardless of how these libraries ultimately appear, the reader who learns Java and object-oriented programming will be ready to take advantage of the new kinds of rapid, component-oriented software development made possible with class libraries.

8.16.1 Example: Queue Abstract Data Type

Each of us stands in line from time to time. A waiting line is also called a *queue*. We wait in line at the supermarket checkout counter, we wait in line to get gasoline, we wait in line to board a bus, we wait in line to pay a toll on the highway, and students know all too well about waiting in line during registration to get the courses they want. Computer systems use many waiting lines internally, so we write programs that simulate what queues are and do.

A queue is a good example of an abstract data type. A queue offers well-understood behavior to its clients. Clients put things in a queue one at a time—using an *enqueue* operation, and the clients get those things back one at a time on demand—using a *dequeue* operation. Conceptually, a queue can become infinitely long. A real queue, of course, is finite. Items are returned from a queue in *first-in, first-out (FIFO)* order—the first item inserted in the queue is the first item removed from the queue.

The queue hides an internal data representation that keeps track of the items currently waiting in line, and it offers a set of operations to its clients, namely *enqueue* and *dequeue*. The clients are not concerned about implementation of the queue. Clients merely want the queue to operate “as advertised.” When a client enqueues a new item, the queue should accept that item and place it internally in some kind of first-in, first-out data structure. When the client wants the next item from the front of the queue, the queue should remove the item from its internal representation and should deliver the item to the outside world in FIFO order (i.e., the item that has been in the queue the longest should be the next one returned by the next *dequeue* operation).

The queue ADT guarantees the integrity of its internal data structure. Clients must not manipulate this data structure directly. Only the queue ADT has access to its internal data (i.e., the queue ADT encapsulates its data). Clients must cause only allowable operations to be performed on the data representation; operations not provided in the ADT’s public interface are rejected by the ADT in some appropriate manner. This could mean issuing an error message, terminating execution, or simply ignoring the operation request.

8.17 (Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulation

In the “Thinking About Objects” sections in Chapters 1 through 7, we introduced the fundamentals of object orientation and developed an object-oriented design for our elevator simulation. In Chapter 8, we introduced the details of programming with Java classes. We now begin implementing our object-oriented design in Java. At the end of this section, we show how to generate code in Java, working from class diagrams. This process is referred to as *forward engineering*.¹

Visibility

Before we begin implementing our design in Java, we apply *member-access modifiers* (see Section 8.2) to the members of our classes. In Chapter 8, we introduced the access specifiers **public** and **private**—these determine the *visibilities* of an object’s attributes and methods to other objects. Before we create class files, we consider which attributes and methods of our classes should be **public** and which should be **private**.



Software Engineering Observation 8.23

Each element of a class should have **private** visibility unless it can be proven that the element needs **public** visibility.

In Chapter 8, we discussed how attributes generally should be **private**, but what about the operations of a class—its methods? These operations are invoked by clients of that class; therefore, the methods normally should be **public**. In the UML, **public** visibility is indicated by placing a plus sign (+) before a particular element (i.e., a method or an attribute); a minus sign (-) indicates **private** visibility. Figure 8.13 shows our updated class diagram with visibility notations included.

1. G. Booch, *The Unified Modeling Language User Guide*. Massachusetts: Addison Wesley Longman, Inc., 1999: 16. [Once code exists, the process of going backward from the code to reproduce design documents is called *reverse engineering*.]

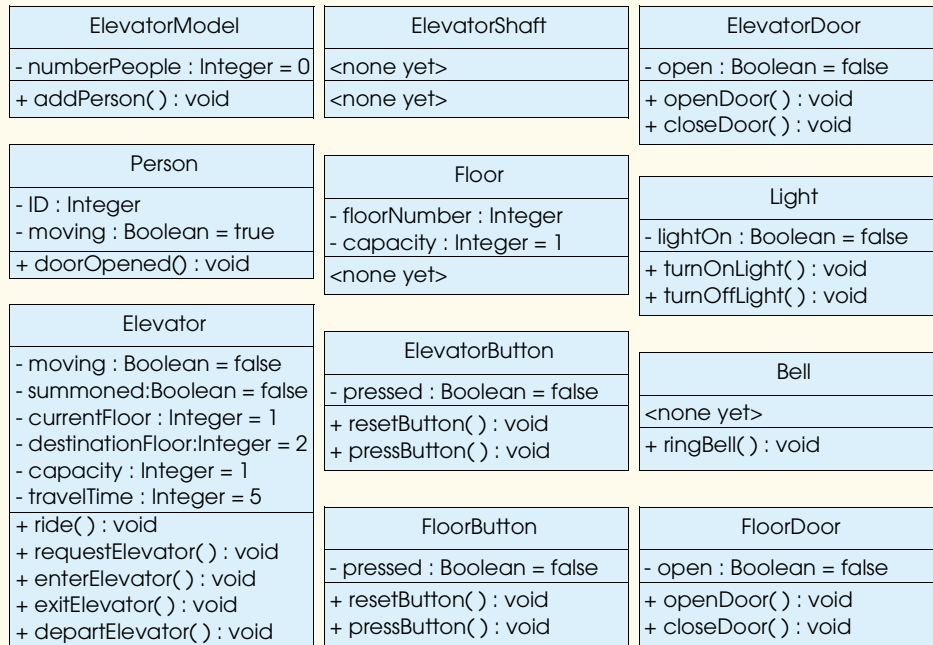


Fig. 8.13 Complete class diagram with visibility notations.

Implementation: Forward Engineering

Forward engineering is the process of transforming a design, such as that in a class diagram, into code of a specific programming language, such as Java. Now that we have discussed programming Java classes, we forward engineer the class diagram of Fig. 8.13 into the Java code for our elevator simulator. The generated code will represent only the “skeleton,” or the structure, of the model.² In Chapters 9 and 10, we modify the code to incorporate inheritance and interfaces, respectively. In Appendix G, Appendix H and Appendix I, we present the complete, working Java code for our model.

As an example, we forward engineer class **Elevator** from Fig. 8.13. We use this figure to determine the attributes and operations of that class. We use the class diagram of Fig. 3.23 to determine associations (and aggregations) among classes. We adhere to the following four guidelines:

1. Use the name located in the first compartment to declare the class as a **public** class with an empty constructor. For example, class **Elevator** yields
2. So far, we have presented only about half of the case-study material—we have not yet discussed inheritance, event handling, multithreading and animation. The standard development process recommends finishing the design process before starting the coding process. Technically, we will not have finished designing our system until we have discussed these additional topics, so our current code implementation might seem premature. We present only a partial implementation illustrating the topics covered in Chapter 8.

```
public class Elevator {
    public Elevator() {}
}
```

2. Use the attributes located in the second compartment to declare the member variables. For example, the **private** attributes **moving**, **summoned**, **currentFloor**, **destinationFloor**, **capacity** and **travelTime** of class **Elevator** yield

```
public class Elevator {
    // class attributes
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // class constructor
    public Elevator() {}
}
```

3. Use the associations described in the class diagram to generate the references to other objects. For example, according to Fig. 3.23, **Elevator** contains one object each of classes **ElevatorDoor**, **ElevatorButton** and **Bell**. This yields

```
public class Elevator {
    // class attributes
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // class objects
    private ElevatorDoor elevatorDoor;
    private ElevatorButton elevatorButton;
    private Bell bell;

    // class constructor
    public Elevator() {}
}
```

4. Use the operations located in the third compartment of Fig. 8.13 to declare the methods. For example, the **public** operations **ride**, **requestElevator**, **enterElevator**, **exitElevator** and **departElevator** in **Elevator** yield

```

public class Elevator {

    // class attributes
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // class objects
    private ElevatorDoor elevatorDoor;
    private ElevatorButton elevatorButton;
    private Bell bell;

    // class constructor
    public Elevator() {}

    // class methods
    public void ride() {}
    public void requestElevator() {}
    public void enterElevator() {}
    public void exitElevator() {}
    public void departElevator() {}
}

```

This concludes the basics of forward engineering. We return to this example at the ends of “Thinking About Objects” Section 9.23 and Section 10.22 to incorporate inheritance, interfaces and event handling.

SUMMARY

- OOP encapsulates data (attributes) and methods (behaviors) into objects; the data and methods of an object are intimately tied together.
- Objects have the property of information hiding. Objects might know how to communicate with one another across well-defined interfaces, but they normally are not allowed to know how other objects are implemented.
- Java programmers concentrate on creating their own user-defined types called classes.
- The non-**static** data components of a class are called instance variables. The **static** data components of a class are called class variables.
- Java uses inheritance to create new classes from existing class definitions.
- Every class in Java is a subclass of **Object**. Thus, every new class definition has the attributes (data) and behaviors (methods) of class **Object**.
- Keywords **public** and **private** are member access modifiers.
- Instance variables and methods declared with member access modifier **public** are accessible wherever the program has a reference to an object of the class in which they are defined.
- Instance variables and methods declared with member access modifier **private** are accessible only to methods of the class in which they are defined.
- Instance variables are normally declared **private** and methods are normally declared **public**.
- The **public** methods (or **public** services) of a class are used by clients of the class to manipulate the data stored in objects of the class.

- A constructor is a method with the exact same name as the class that initializes the instance variables of an object of the class when the object is instantiated. Constructor methods can be overloaded for a class. Constructors can take arguments but cannot specify a return value type.
- Constructors and other methods that change instance variable values should always maintain objects in a consistent state.
- Method **toString** takes no arguments and returns a **String**. The original **toString** method of class **Object** is a placeholder that is normally redefined by a subclass.
- When an object is instantiated, operator **new** allocates the memory for the object, then **new** calls the constructor for the class to initialize the instance variables of the object.
- If the **.class** files for the classes used in a program are in the same directory as the class that uses them, **import** statements are not required.
- Concatenating a **String** and any object results in an implicit call to the object's **toString** method to convert the object to a **String**, then the **Strings** are concatenated.
- Within a class's scope, class members are accessible to all of that class's methods and can be referenced simply by name. Outside a class's scope, class members can only be accessed off a "handle" (i.e., a reference to an object of the class).
- If a method defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the method-scope variable in the method. A hidden instance variable can be accessed in the method by preceding its name with the keyword **this** and the dot operator.
- Each class and interface in the Java API belongs to a specific package that contains a group of related classes and interfaces.
- Packages are actually directory structures used to organize classes and interfaces. Packages provide a mechanism for software reuse and a convention for unique class names.
- Creating a reusable class requires: defining a **public** class, adding a **package** statement to the class definition file, compiling the class into the appropriate package directory structure and importing the class into a program.
- When compiling a class in a package, the option **-d** must be passed to the compiler to specify where to create (or locate) all the directories in the **package** statement.
- The **package** directory names become part of the class name when the class is compiled. Use this fully qualified name in programs or **import** the class and use its short name (the name of the class by itself) in the program.
- If no constructors are defined for a class, the compiler creates a default constructor.
- When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods.
- Classes often provide **public** methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) **private** instance variables. *Get* methods are also commonly called accessor methods or query methods. *Set* methods are also commonly called mutator methods (because they typically change a value).
- Every event has a source—the GUI component with which the user interacted to signal the program to do a task.
- Use the keyword **final** to specify that a variable is not modifiable and that any attempt to modify the variable is an error. A **final** variable cannot be modified by assignment after it is initialized. Such a variable must be initialized in its declaration or in every constructor of the class.
- With composition, a class has references to objects of other classes as members.
- When no member access modifier is provided for a method or variable when it is defined in a class, the method or variable is considered to have package access.

- If a program uses multiple classes from the same package, these classes can access each other's package-access methods and data directly through a reference to an object.
- Each object has access to a reference to itself called the **this** reference that can be used inside the methods of the class to refer to the object's data and other methods explicitly.
- Any time you have a reference in a program (even as the result of a method call), the reference can be followed by a dot operator and a call to one of the methods for the reference type.
- Java performs automatic garbage collection of memory. When an object is no longer used in the program (i.e., there are no references to the object), the object is marked for garbage collection.
- Every class in Java can have a finalizer method that returns resources to the system. A class's finalizer method always has the name **finalize**, receives no parameters and returns no value. Method **finalize** is originally defined in class **Object** as a placeholder that does nothing. This guarantees that every class has a **finalize** method for the garbage collector to call.
- A **static** class variable represents class-wide information—all objects of the class share the same piece of data. A class's **public static** members can be accessed through a reference to any object of that class, or they can be accessed through the class name using the dot operator.
- **public static** method **gc** from class **System** suggests that the garbage collector immediately make a best effort attempt to collect garbage objects. The garbage collector is not guaranteed to collect objects in a specific order.
- A method declared **static** cannot access **nonstatic** class members. Unlike **nonstatic** methods, a **static** method has no **this** reference, because **static** class variables and **static** class methods exist independent of any objects of a class.
- **static** class members exist even when no objects of that class exist—they are available as soon as the class is loaded into memory at execution time.

TERMINOLOGY

abstract data type (ADT)	extends
access method	extensibility
aggregation	finalizer
attribute	get method
behavior	helper method
cascaded method calls	implementation of a class
class	information hiding
class definition	initialize a class object
class library	instance method
class method (static)	instance of a class
class scope	instance variable
class variable	instantiate an object of a class
client of a class	interface to a class
composition	member access control
concatenated method calls	member access modifiers
consistent state for an instance variable	member access operator (.)
constructor	message
container class	method
-d compiler option	method calls
data type	mutator method
default constructor	new operator
dot operator (.)	no-argument constructor
encapsulation	object

object-based programming (OBP)
 object-oriented programming (OOP)
 package access
package statement
 predicate method
 principle of least privilege
private
 programmer-defined type
public
 public interface of a class
 query method

rapid applications development (RAD)
 reusable code
 services of a class
 set method
 software reusability
 static class variable
 static method
this reference
 user-defined type
 utility method

SELF-REVIEW EXERCISES

8.1 Fill in the blanks in each of the following statements:

- Class members are accessed via the _____ operator in conjunction with a reference to an object of the class.
- Members of a class specified as _____ are accessible only to methods of the class.
- A _____ is a special method used to initialize the instance variables of a class.
- A _____ method is used to assign values to **private** instance variables of a class.
- Methods of a class are normally made _____ and instance variables of a class are normally made _____.
- A _____ method is used to retrieve values of **private** data of a class.
- The keyword _____ introduces a class definition.
- Members of a class specified as _____ are accessible anywhere an object of the class is in scope.
- The _____ operator dynamically allocates memory for an object of a specified type and returns a _____ to that type.
- A _____ instance variable represents class-wide information.
- The keyword _____ specifies that an object or variable is not modifiable after it is initialized.
- A method declared **static** cannot access _____ class members.

ANSWERS TO SELF-REVIEW EXERCISES

8.1 a) dot (.). b) **private**. c) constructor. d) set. e) **public, private**. f) get. g) **class**. h) **public**. i) **new**, reference. j) **static**. k) **final**. l) **nonstatic**.

EXERCISES

8.2 Create a class called **Complex** for performing arithmetic with complex numbers. Write a driver program to test your class.

Complex numbers have the form

$$\text{realPart} + \text{imaginaryPart} * i$$

where i is

$$\sqrt{-1}$$

Use floating-point variables to represent the **private** data of the class. Provide a constructor method that enables an object of this class to be initialized when it is declared. Provide a no-arg-

ment constructor with default values in case no initializers are provided. Provide **public** methods for each of the following:

- a) Addition of two **Complex** numbers: The real parts are added together and the imaginary parts are added together.
- b) Subtraction of two **Complex** numbers: The real part of the right operand is subtracted from the real part of the left operand and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.
- c) Printing **Complex** numbers in the form **(a, b)**, where **a** is the real part and **b** is the imaginary part.

8.3 Create a class called **Rational** for performing arithmetic with fractions. Write a driver program to test your class.

Use integer variables to represent the **private** instance variables of the class—the **numerator** and the **denominator**. Provide a constructor method that enables an object of this class to be initialized when it is declared. The constructor should store the fraction in reduced form (i.e., the fraction

2/4

would be stored in the object as 1 in the **numerator** and 2 in the **denominator**). Provide a no-argument constructor with default values in case no initializers are provided. Provide **public** methods for each of the following:

- a) Addition of two **Rational** numbers. The result of the addition should be stored in reduced form.
- b) Subtraction of two **Rational** numbers. The result of the subtraction should be stored in reduced form.
- c) Multiplication of two **Rational** numbers. The result of the multiplication should be stored in reduced form.
- d) Division of two **Rational** numbers. The result of the division should be stored in reduced form.
- e) Printing **Rational** numbers in the form **a/b**, where **a** is the **numerator** and **b** is the **denominator**.
- f) Printing **Rational** numbers in floating-point format. (Consider providing formatting capabilities that enable the user of the class to specify the number of digits of precision to the right of the decimal point.)

8.4 Modify the **Time3** class of Fig. 8.8 to include the **tick** method that increments the time stored in a **Time3** object by one second. Also provide method **incrementMinute** to increment the minute and method **incrementHour** to increment the hour. The **Time3** object should always remain in a consistent state. Write a driver program that tests the **tick** method, the **incrementMinute** method and the **incrementHour** method to ensure that they work correctly. Be sure to test the following cases:

- a) incrementing into the next minute.
- b) incrementing into the next hour.
- c) incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

8.5 Modify the **Date** class of Fig. 8.13 to perform error-checking on the initializer values for instance variables **month**, **day** and **year** (currently it validates only the month and day). Also, provide a method **nextDay** to increment the day by one. The **Date** object should always remain in a consistent state. Write a driver program that tests the **nextDay** method in a loop that prints the date during each iteration of the loop to illustrate that the **nextDay** method works correctly. Be sure to test the following cases:

- a) incrementing into the next month.
- b) incrementing into the next year.

8.6 Combine the modified **Time3** class of Exercise 8.5 and the modified **Date** class of Exercise 8.5 into one class called **DateAndTime**. Modify the **tick** method to call the **nextDay** method if the time is incremented into the next day. Modify methods **toString** and **toUniversalString()** to output the date in addition to the time. Write a driver program to test the new class **DateAndTime**. Specifically test incrementing the time to the next day.

8.7 Modify the set methods in class **Time3** of Fig. 8.8 to return appropriate error values if an attempt is made to set one of the instance variables **hour**, **minute** or **second** of an object of class **Time** to an invalid value. (*Hint:* Use **boolean** return types on each method.)

8.8 Create a class **Rectangle**. The class has attributes **length** and **width**, each of which defaults to 1. It has methods that calculate the **perimeter** and the **area** of the rectangle. It has *set* and *get* methods for both **length** and **width**. The *set* methods should verify that **length** and **width** are each floating-point numbers larger than 0.0 and less than 20.0. Write a program to test class **Rectangle**.

8.9 Create a more sophisticated **Rectangle** class than the one you created in Exercise 8.8. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a *set* method that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single *x*- or *y*-coordinate larger than 20.0. The *set* method also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide methods to calculate the **length**, **width**, **perimeter** and **area**. The length is the larger of the two dimensions. Include a predicate method **isSquare** which determines whether the rectangle is a square. Write a program to test class **Rectangle**.

8.10 Modify the **Rectangle** class of Exercise 8.9 to include a **draw** method that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Use the methods of the **Graphics** class to help output the **Rectangle**. If you feel ambitious, you might include methods to scale the size of the rectangle, rotate it and move it around within the designated portion of the first quadrant.

8.11 Create a class **HugeInteger** which uses a 40-element array of digits to store integers as large as 40 digits each. Provide methods **inputHugeInteger**, **outputHugeInteger**, **addHugeIntegers** and **subtractHugeIntegers**. For comparing **HugeInteger** objects, provide methods **isEqualTo**, **isNotEqualTo**, **isGreaterThan**, **isLessThan**, **isGreaterThanOrEqualTo** and **isLessThanOrEqualTo**—each of these is a “predicate” method that simply returns **true** if the relationship holds between the two **HugeIntegers** and returns **false** if the relationship does not hold. Provide a predicate method **isZero**. If you feel ambitious, also provide the method **multiplyHugeIntegers**, the method **divideHugeIntegers** and the method **modulusHugeIntegers**.

8.12 Create a class **TicTacToe** that will enable you to write a complete program to play the game of Tic-Tac-Toe. The class contains as private data a 3-by-3 double array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square; place a 2 wherever the second player moves. Each move must be to an empty square. After each move determine whether the game has been won and whether the game is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players automatically. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional Tic-Tac-Toe on a 4-by-4-by-4 board [Note: This is a challenging project that could take many weeks of effort!].

8.13 Explain the notion of package access in Java. Explain the negative aspects of package access as described in the text.

8.14 What happens when a return type, even **void**, is specified for a constructor?

8.15 Create a **Date** class with the following capabilities:

- a) Output the date in multiple formats such as

```
MM/DD/YYYY
June 14, 1992
DDD YYYY
```

- b) Use overloaded constructors to create **Date** objects initialized with dates of the formats in part a).

[Hint: You can compare **Strings** using method **equals**. Suppose you have two **String** references **s1** and **s2**, if those **Strings** are equal, **s1.equals(s2)** returns **true**; otherwise, it returns **false**.]

8.16 Create class **SavingsAccount**. Use a **static** class variable to store the **annualInterestRate** for all account holders. Each object of the class contains a **private** instance variable **savingsBalance** indicating the amount the saver currently has on deposit. Provide method **calculateMonthlyInterest** to calculate the monthly interest by multiplying the **savingsBalance** by **annualInterestRate** divided by 12; this interest should be added to **savingsBalance**. Provide a **static** method **modifyInterestRate** that sets the **annualInterestRate** to a new value. Write a driver program to test class **SavingsAccount**. Instantiate two **savingsAccount** objects, **saver1** and **saver2**, with balances of \$2000.00 and \$3000.00, respectively. Set **annualInterestRate** to 4%, then calculate the monthly interest and print the new balances for each of the savers. Then set the **annualInterestRate** to 5% and calculate the next month's interest and print the new balances for each of the savers.

8.17 Create class **IntegerSet**. Each object of the class can hold integers in the range 0 through 100. A set is represented internally as an array of **booleans**. Array element **a[i]** is **true** if integer *i* is in the set. Array element **a[j]** is **false** if integer *j* is not in the set. The no-argument constructor initializes a set to the so-called "empty set" (i.e., a set whose array representation contains all **false** values).

Provide the following methods: Method **unionOfIntegerSets** creates a third set which is the set-theoretic union of two existing sets (i.e., an element of the third set's array is set to **true** if that element is **true** in either or both of the existing sets; otherwise, the element of the third set is set to **false**). Method **intersectionOfIntegerSets** creates a third set which is the set-theoretic intersection of two existing sets i.e., an element of the third set's array is set to **false** if that element is **false** in either or both of the existing sets; otherwise, the element of the third set is set to **true**). Method **insertElement** inserts a new integer *k* into a set (by setting **a[k]** to **true**). Method **deleteElement** deletes integer *m* (by setting **a[m]** to **false**). Method **setPrint** prints a set as a list of numbers separated by spaces. Print only those elements that are present in the set. Print --- for an empty set. Method **isEqualTo** determines if two sets are equal. Write a program to test your **IntegerSet** class. Instantiate several **IntegerSet** objects. Test that all your methods work properly.

8.18 It would be perfectly reasonable for the **Time1** class of Figure 8.1 to represent the time internally as the number of seconds since midnight rather than the three integer values **hour**, **minute** and **second**. Clients could use the same **public** methods and get the same results. Modify the **Time1** class of Figure 8.1 to implement the **Time1** as the number of seconds since midnight and show that there is no change visible to the clients of the class.

8.19 (*Drawing Program*) Create a drawing applet that randomly draws lines, rectangles and ovals. For this purpose, create a set of "smart" shape classes where objects of these classes know how to draw themselves if provided with a **Graphics** object that tells them where to draw (i.e., the applet's **Graphics** object allows a shape to draw on the applet's background). The class names should be **MyLine**, **MyRect** and **MyOval**.

The data for class **MyLine** should include *x1*, *y1*, *x2* and *y2* coordinates. Method **drawLine** method of class **Graphics** will connect the two points supplied with a line. The data for classes **MyRect** and **MyOval** should include an upper-left *x*-coordinate value, an upper-left *y*-coordinate value, a *width* (must be nonnegative) and a *height* (must be nonnegative). All data in each class must be **private**.

In addition to the data, each class should define at least the following **public** methods:

- a) A constructor with no arguments that sets the coordinates to 0.
- b) A constructor with arguments that sets the coordinates to the supplied values.
- c) Set methods for each individual piece of data that allow the programmer to independently set any piece of data in a shape (e.g., if you have an instance variable **x1**, you should have a method **setX1**).
- d) Get methods for each individual piece of data that allow the programmer to independently retrieve any piece of data in a shape (e.g., if you have an instance variable **x1**, you should have a method **getX1**).
- e) A **draw** method with the first line

```
public void draw( Graphics g )
```

will be called from the applet's **paint** method to draw a shape onto the screen.

The preceding methods are required. If you would like to provide more methods for flexibility, please do so.

Begin by defining class **MyLine** and an applet to test your classes. The applet should have a **MyLine** instance variable **line** that can refer to one **MyLine** object (created in the applet's **init** method with random coordinates). The applet's **paint** method should draw the shape with a statement like

```
line.draw( g );
```

where **line** is the **MyLine** reference and **g** is the **Graphics** object that the shape will use to draw itself on the applet.

Next, change the single **MyLine** reference into an array of **MyLine** references and hard code several **MyLine** objects into the program for drawing. The applet's **paint** method should walk through the array of **MyLine** objects and draw every one.

After the preceding part is working, you should define the **MyOval** and **MyRect** classes and add objects of these classes into the **MyRect** and **MyOval** arrays. The applet's **paint** method should walk through each array and draw every shape. Create five shapes of each type.

Once the applet is running, select **Reload** from the **appletviewer**'s **Applet** menu to reload the applet. This will cause the applet to choose new random numbers for the shapes and draw the shapes again.

In Chapter 9, we will modify this exercise to take advantage of the similarities between the classes and to avoid reinventing the wheel.