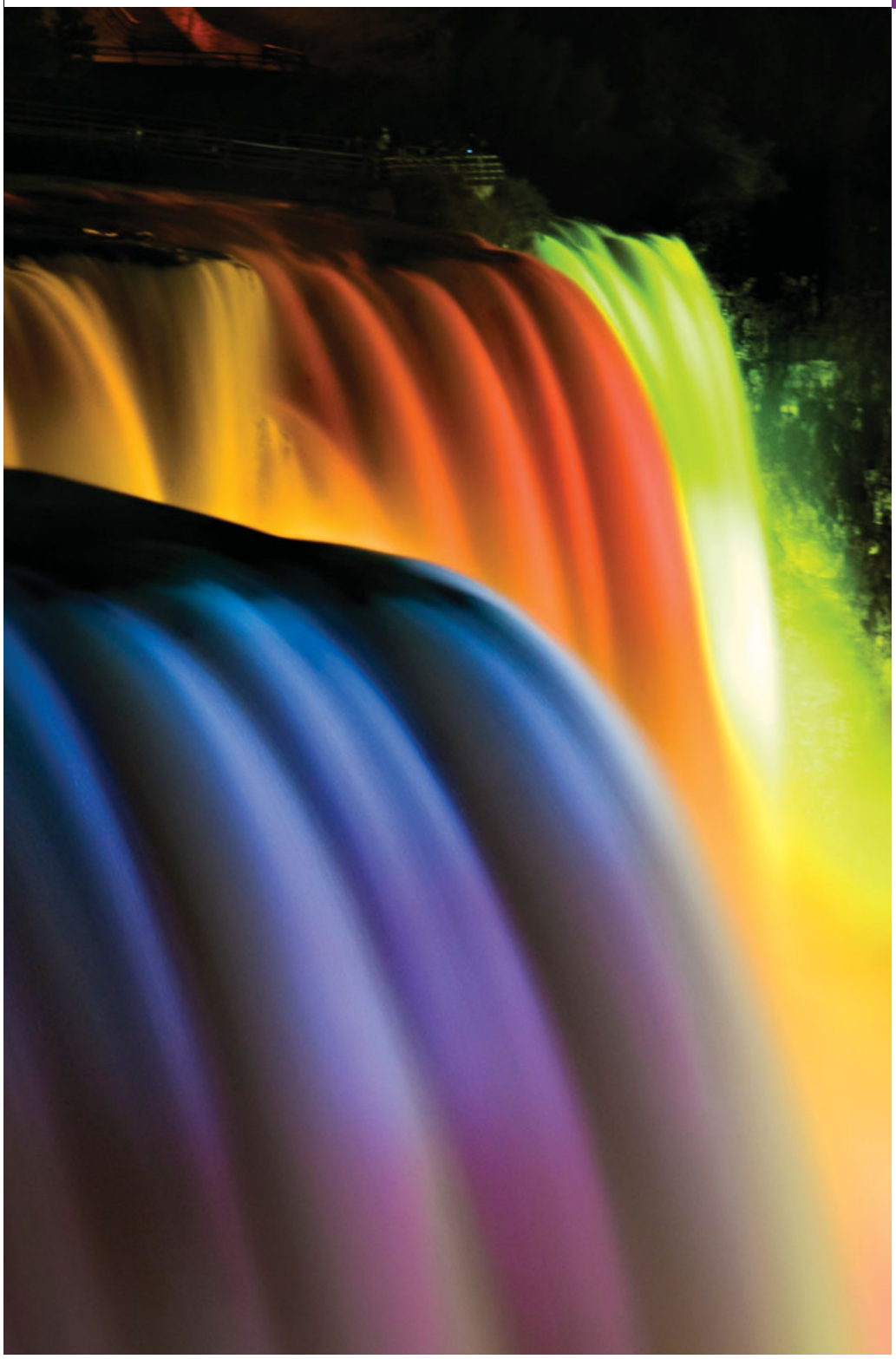
8

Classes and Objects: A Deeper Look



Instead of this absurd division into sexes, they ought to class people as static and dynamic.
—Evelyn Waugh

Is it a world to hide virtues in?
—William Shakespeare

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

Don't be "consistent," but be simply true.
—Oliver Wendell Holmes, Jr.

Objectives

In this chapter you'll learn:

- Encapsulation and data hiding.
- To use keyword `this`.
- To use `static` variables and methods.
- To import `static` members of a class.
- To use the `enum` type to create sets of constants with unique identifiers.
- To declare `enum` constants with parameters.
- To organize classes in packages to promote reuse.

8.1 Introduction

We now take a deeper look at building classes, controlling access to members of a class and creating constructors. We discuss composition—a capability that allows a class to have references to objects of other classes as members. We reexamine the use of *set* and *get* methods. Recall that Section 6.10 introduced the basic `enum` type to declare a set of constants. In this chapter, we discuss the relationship between `enum` types and classes, demonstrating that an `enum`, like a class, can be declared in its own file with constructors, methods and fields. The chapter also discusses `static` class members and `final` instance variables in detail. Finally, we explain how to organize classes in packages to help manage large applications and promote reuse, then show a special relationship between classes in the same package.

8.2 `Time` Class Case Study

Our first example consists of two classes—`Time1` (Fig. 8.1) and `Time1Test` (Fig. 8.2). Class `Time1` represents the time of day. Class `Time1Test` is an application class in which the `main` method creates one object of class `Time1` and invokes its methods. These classes must be declared in *separate* files because they're both `public` classes. The output of this program appears in Fig. 8.2.

`Time1` Class Declaration

Class `Time1`'s `private int` instance variables `hour`, `minute` and `second` (Fig. 8.1, lines 6–8) represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23). Class `Time1` contains `public` methods `setTime` (lines 12–25), `toUniversalString` (lines 28–31) and `toString` (lines 34–39). These methods are also called the **`public` services** or the **`public` interface** that the class provides to its clients.

Default Constructor

In this example, class `Time1` does not declare a constructor, so the class has a default constructor that's supplied by the compiler. Each instance variable implicitly receives the

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime( int h, int m, int s )
13    {
14       // validate hour, minute and second
15       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
16          ( s >= 0 && s < 60 ) )
17       {
18          hour = h;
19          minute = m;
20          second = s;
21       } // end if
22       else
23          throw new IllegalArgumentException(
24             "hour, minute and/or second was out of range" );
25    } // end method setTime
26
27    // convert to String in universal-time format (HH:MM:SS)
28    public String toUniversalString()
29    {
30       return String.format( "%02d:%02d:%02d", hour, minute, second );
31    } // end method toUniversalString
32
33    // convert to String in standard-time format (H:MM:SS AM or PM)
34    public String toString()
35    {
36       return String.format( "%d:%02d:%02d %s",
37          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
38          minute, second, ( hour < 12 ? "AM" : "PM" ) );
39    } // end method toString
40 } // end class Time1
```

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format.

default value 0 for an `int`. Instance variables also can be initialized when they're declared in the class body, using the same initialization syntax as with a local variable.

Method `setTime` *and Throwing Exceptions*

Method `setTime` (lines 12–25) is a `public` method that declares three `int` parameters and uses them to set the time. Lines 15–16 test each argument to determine whether the value is in the proper range, and, if so, lines 18–20 assign the values to the `hour`, `minute` and `second` instance variables. The `hour` value must be greater than or equal to 0 and less than 24, because universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both

minute and second values must be greater than or equal to 0 and less than 60. For values outside these ranges, SetTime **throws an exception** of type **IllegalArgumentException** (lines 23–24), which notifies the client code that an invalid argument was passed to the method. As you learned in Chapter 7, you can use try...catch to catch exceptions and attempt to recover from them, which we'll do in Fig. 8.2. The **throw statement** (line 23) creates a new object of type IllegalArgumentException. The parentheses following the class name indicate a call to the IllegalArgumentException constructor. In this case, we call the constructor that allows us to specify a custom error message. After the exception object is created, the throw statement immediately terminates method setTime and the exception is returned to the code that attempted to set the time.

Method `toUniversalString`

Method toUniversalString (lines 28–31) takes no arguments and returns a String in universal-time format, consisting of two digits each for the hour, minute and second. For example, if the time were 1:30:07 PM, the method would return 13:30:07. Line 22 uses static method **format** of class String to return a String containing the formatted hour, minute and second values, each with two digits and possibly a leading 0 (specified with the 0 flag). Method format is similar to method System.out.printf except that format *returns* a formatted String rather than displaying it in a command window. The formatted String is returned by method toUniversalString.

Method `toString`

Method toString (lines 34–39) takes no arguments and returns a String in standard-time format, consisting of the hour, minute and second values separated by colons and followed by AM or PM (e.g., 1:27:06 PM). Like method toUniversalString, method toString uses static String method format to format the minute and second as two-digit values, with leading zeros if necessary. Line 29 uses a conditional operator (?:) to determine the value for hour in the String—if the hour is 0 or 12 (AM or PM), it appears as 12; otherwise, it appears as a value from 1 to 11. The conditional operator in line 30 determines whether AM or PM will be returned as part of the String.

Recall from Section 6.4 that all objects in Java have a toString method that returns a String representation of the object. We chose to return a String containing the time in standard-time format. Method toString is called implicitly whenever a Time1 object appears in the code where a String is needed, such as the value to output with a %s format specifier in a call to System.out.printf.

Using Class `Time1`

As you learned in Chapter 3, each class you declare represents a new *type* in Java. Therefore, after declaring class Time1, we can use it as a type in declarations such as

```
Time1 sunset; // sunset can hold a reference to a Time1 object
```

The Time1Test application class (Fig. 8.2) uses class Time1. Line 9 declares and creates a Time1 object and assigns it to local variable time. Operator new implicitly invokes class Time1's default constructor, since Time1 does not declare any constructors. Lines 12–16 output the time first in universal-time format (by invoking time's toUniversalString method in line 13), then in standard-time format (by explicitly invoking time's toString method in line 15) to confirm that the Time1 object was initialized properly. Next, line 19

invokes method `setTime` of the `time` object to change the time. Then lines 20–24 output the time again in both formats to confirm that it was set correctly.

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an application.
3
4  public class Time1Test
5  {
6     public static void main( String[] args )
7     {
8        // create and initialize a Time1 object
9        Time1 time = new Time1(); // invokes Time1 constructor
10
11       // output string representations of the time
12       System.out.print( "The initial universal time is: " );
13       System.out.println( time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println( time.toString() );
16       System.out.println(); // output a blank line
17
18       // change time and output updated time
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println( time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println( time.toString() );
24       System.out.println(); // output a blank line
25
26       // attempt to set time with invalid values
27       try
28       {
29          time.setTime( 99, 99, 99 ); // all values out of range
30       } // end try
31       catch ( IllegalArgumentException e )
32       {
33          System.out.printf( "Exception: %s\n\n", e.getMessage() );
34       } // end catch
35
36       // display time after attempt to set invalid values
37       System.out.println( "After attempting invalid settings:" );
38       System.out.print( "Universal time: " );
39       System.out.println( time.toUniversalString() );
40       System.out.print( "Standard time: " );
41       System.out.println( time.toString() );
42    } // end main
43 } // end class Time1Test
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM
```

Fig. 8.2 | `Time1` object used in an application. (Part 1 of 2.)

```
Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.2 | `Time1` object used in an application. (Part 2 of 2.)

Calling `Time1` Method `setTime` with Invalid Values

To illustrate that method `setTime` validates its arguments, line 29 calls method `setTime` with invalid arguments of `99` for the `hour`, `minute` and `second`. This statement is placed in a `try` block (lines 27–30) in case `setTime` throws an `IllegalArgumentException`, which it will do since the arguments are all invalid. When this occurs, the exception is caught at lines 31–34, and line 33 displays the exception's error message by calling its `getMessage` method. Lines 37–41 output the time again in both formats to confirm that `setTime` did not change the time when invalid arguments were supplied.

Notes on the `Time1` Class Declaration

Consider several issues of class design with respect to class `Time1`. The instance variables `hour`, `minute` and `second` are each declared `private`. 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 `Time1` to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight. Clients could use the same `public` methods and get the same results without being aware of this. (Exercise 8.5 asks you to represent the time in class `Time1` as the number of seconds since midnight and show that indeed no change is visible to the clients of the class.)

Software Engineering Observation 8.1

Classes simplify programming, because the client can use only the `public` methods exposed by the class. Such methods are usually client oriented rather than implementation oriented. Clients are neither aware of, nor involved in, a class's implementation. Clients generally care about what *the class does but not* how *the class does it.*

Software Engineering Observation 8.2

Interfaces change less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation reduces the possibility that other program parts will become dependent on class implementation details.

8.3 Controlling Access to Members

The access modifiers `public` and `private` control access to a class's variables and methods. In Chapter 9, we'll introduce the additional access modifier `protected`. As we stated in Section 8.2, the primary purpose of `public` methods is to present to the class's clients a view of the services the class provides (the class's `public` interface). Clients need not be concerned with how the class accomplishes its tasks. For this reason, the class's `private` variables and `private` methods (i.e., its implementation details) are *not* accessible to its clients.

Figure 8.3 demonstrates that `private` class members are not accessible outside the class. Lines 9–11 attempt to access directly the `private` instance variables `hour`, `minute` and `second` of the `Time1` object `time`. When this program is compiled, the compiler generates error messages that these `private` members are not accessible. This program assumes that the `Time1` class from Fig. 8.1 is used.

Common Programming Error 8.1

An attempt by a method that's not a member of a class to access a `private` member of that class is a compilation error.

```
1  // Fig. 8.3: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest
4  {
5     public static void main( String[] args )
6     {
7        Time1 time = new Time1(); // create and initialize Time1 object
8
9        time.hour = 7; // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    } // end main
13 } // end class MemberAccessTest
```

```
MemberAccessTest.java:9: hour has private access in Time1
      time.hour = 7; // error: hour has private access in Time1
           ^
MemberAccessTest.java:10: minute has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
           ^
MemberAccessTest.java:11: second has private access in Time1
      time.second = 30; // error: second has private access in Time1
           ^
3 errors
```

Fig. 8.3 | Private members of class `Time1` are not accessible.

8.4 Referring to the Current Object's Members with the `this` Reference

Every object can access a reference to itself with keyword **`this`** (sometimes called the **`this` reference**). When a non-`static` method is called for a particular object, the method's body implicitly uses keyword `this` to refer to the object's instance variables and other methods. This enables the class's code to know which object should be manipulated. As you'll see in Fig. 8.4, you can also use keyword `this` explicitly in a non-`static` method's body. Section 8.5 shows another interesting use of keyword `this`. Section 8.11 explains why keyword `this` cannot be used in a `static` method.

We now demonstrate implicit and explicit use of the `this` reference (Fig. 8.4). This example is the first in which we declare *two* classes in one file—class `ThisTest` is declared

in lines 4–11, and class `SimpleTime` in lines 14–47. We do this to demonstrate that when you compile a `.java` file containing more than one class, the compiler produces a separate class file with the `.class` extension for every compiled class. In this case, two separate files are produced—`SimpleTime.class` and `ThisTest.class`. When one source-code (`.java`) file contains multiple class declarations, the compiler places both class files for those classes in the same directory. Note also in Fig. 8.4 that only class `ThisTest` is declared `public`. A source-code file can contain only one `public` class—otherwise, a compilation error occurs. Non-`public` classes can be used only by other classes in the same package. So, in this example, class `SimpleTime` can be used only by class `ThisTest`.

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest
5  {
6     public static void main( String[] args )
7     {
8        SimpleTime time = new SimpleTime( 15, 30, 19 );
9        System.out.println( time.buildString() );
10    } // end main
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16    private int hour; // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
20    // if the constructor uses parameter names identical to
21    // instance variable names the "this" reference is
22    // required to distinguish between the names
23    public SimpleTime( int hour, int minute, int second )
24    {
25       this.hour = hour; // set "this" object's hour
26       this.minute = minute; // set "this" object's minute
27       this.second = second; // set "this" object's second
28    } // end SimpleTime constructor
29
30    // use explicit and implicit "this" to call toUniversalString
31    public String buildString()
32    {
33       return String.format( "%24s: %s\n%24s: %s",
34          "this.toUniversalString()", this.toUniversalString(),
35          "toUniversalString()", toUniversalString() );
36    } // end method buildString
37
38    // convert to String in universal-time format (HH:MM:SS)
39    public String toUniversalString()
40    {
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 2.)

```
41         // "this" is not required here to access instance variables,
42         // because method does not have local variables with same
43         // names as instance variables
44         return String.format( "%02d:%02d:%02d",
45            this.hour, this.minute, this.second );
46      } // end method toUniversalString
47   } // end class SimpleTime
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 2 of 2.)

Class `SimpleTime` (lines 14–47) declares three `private` instance variables—`hour`, `minute` and `second` (lines 16–18). The constructor (lines 23–28) receives three `int` arguments to initialize a `SimpleTime` object. We used parameter names for the constructor (line 23) that are identical to the class's instance-variable names (lines 16–18). We don't recommend this practice, but we did it here to shadow (hide) the corresponding instance variables so that we could illustrate a case in which *explicit* use of the `this` reference is required. If a method contains a local variable with the *same* name as a field, that method will refer to the local variable rather than the field. In this case, the local variable shadows the field in the method's scope. However, the method can use the `this` reference to refer to the shadowed field explicitly, as shown on the left sides of the assignments in lines 25–27 for `SimpleTime`'s shadowed instance variables.

Method `buildString` (lines 31–36) returns a `String` created by a statement that uses the `this` reference explicitly and implicitly. Line 34 uses it explicitly to call method `toUniversalString`. Line 35 uses it implicitly to call the same method. Both lines perform the same task. You typically will not use `this` explicitly to reference other methods within the current object. Also, line 45 in method `toUniversalString` explicitly uses the `this` reference to access each instance variable. This is *not* necessary here, because the method does *not* have any local variables that shadow the instance variables of the class.

Common Programming Error 8.2

It's often a logic error when a method contains a parameter or local variable that has the same name as a field of the class. In this case, use reference `this` if you wish to access the field of the class—otherwise, the method parameter or local variable will be referenced.

Error-Prevention Tip 8.1

Avoid method-parameter names or local-variable names that conflict with field names. This helps prevent subtle, hard-to-locate bugs.

Performance Tip 8.1

Java conserves storage by maintaining only one copy of each method per class—this method is invoked by every *object of the class. Each object, on the other hand, has its own copy of the class's instance variables (i.e., non-`static` fields). Each method of the class implicitly uses `this` to determine the specific object of the class to manipulate.*

Application class `ThisTest` (lines 4–11) demonstrates class `SimpleTime`. Line 8 creates an instance of class `SimpleTime` and invokes its constructor. Line 9 invokes the object's `buildString` method, then displays the results.

8.5 `Time` Class Case Study: Overloaded Constructors

As you know, you can declare your own constructor to specify how objects of a class should be initialized. Next, we demonstrate a class with several **overloaded constructors** that enable objects of that class to be initialized in different ways. To overload constructors, simply provide multiple constructor declarations with different signatures.

Class `Time2` with Overloaded Constructors

The default constructor for class `Time1` (Fig. 8.1) initialized `hour`, `minute` and `second` to their default `0` values (which is midnight in universal time). The default constructor does not enable the class's clients to initialize the time with specific nonzero values. Class `Time2` (Fig. 8.5) contains five overloaded constructors that provide convenient ways to initialize objects of the new class `Time2`. Each constructor initializes the object to begin in a consistent state. In this program, four of the constructors invoke a fifth, which in turn calls method `setTime` to ensure that the value supplied for `hour` is in the range 0 to 23, and the values for `minute` and `second` are each in the range 0 to 59. The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration. Class `Time2` also provides *set* and *get* methods for each instance variable.

```
1  // Fig. 8.5: Time2.java
2  // Time2 class declaration with overloaded constructors.
3
4  public class Time2
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // Time2 no-argument constructor:
11    // initializes each instance variable to zero
12    public Time2()
13    {
14       this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
15    } // end Time2 no-argument constructor
16
17    // Time2 constructor: hour supplied, minute and second defaulted to 0
18    public Time2( int h )
19    {
20       this( h, 0, 0 ); // invoke Time2 constructor with three arguments
21    } // end Time2 one-argument constructor
22
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 1 of 3.)

```
23    // Time2 constructor: hour and minute supplied, second defaulted to 0
24    public Time2( int h, int m )
25    {
26       this( h, m, 0 ); // invoke Time2 constructor with three arguments
27    } // end Time2 two-argument constructor
28
29    // Time2 constructor: hour, minute and second supplied
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoke setTime to validate time
33    } // end Time2 three-argument constructor
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 three-argument constructor
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // end Time2 constructor with a Time2 object argument
41
42    // Set Methods
43    // set a new time value using universal time;
44    // validate the data
45    public void setTime( int h, int m, int s )
46    {
47       setHour( h ); // set the hour
48       setMinute( m ); // set the minute
49       setSecond( s ); // set the second
50    } // end method setTime
51
52    // validate and set hour
53    public void setHour( int h )
54    {
55       if ( h >= 0 && h < 24 )
56          hour = h;
57       else
58          throw new IllegalArgumentException( "hour must be 0-23" );
59    } // end method setHour
60
61    // validate and set minute
62    public void setMinute( int m )
63    {
64       if ( m >= 0 && m < 60 )
65          minute = m;
66       else
67          throw new IllegalArgumentException( "minute must be 0-59" );
68    } // end method setMinute
69
70    // validate and set second
71    public void setSecond( int s )
72    {
73       if ( s >= 0 && s < 60 )
74          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 2 of 3.)

```
75          else
76             throw new IllegalArgumentException( "second must be 0-59" );
77       } // end method setSecond
78
79       // Get Methods
80       // get hour value
81       public int getHour()
82       {
83          return hour;
84       } // end method getHour
85
86       // get minute value
87       public int getMinute()
88       {
89          return minute;
90       } // end method getMinute
91
92       // get second value
93       public int getSecond()
94       {
95          return second;
96       } // end method getSecond
97
98       // convert to String in universal-time format (HH:MM:SS)
99       public String toUniversalString()
100      {
101         return String.format(
102            "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
103      } // end method toUniversalString
104
105      // convert to String in standard-time format (H:MM:SS AM or PM)
106      public String toString()
107      {
108         return String.format( "%d:%02d:%02d %s",
109            ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
110            getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
111      } // end method toString
112   } // end class Time2
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 3 of 3.)

Class `Time2`'s Constructors

Lines 12–15 declare a so-called **no-argument constructor** that's invoked without arguments. Once you declare any constructors in a class, the compiler will *not* provide a default constructor. This no-argument constructor ensures that class `Time2`'s clients can create `Time2` objects with default values. Such a constructor simply initializes the object as specified in the constructor's body. In the body, we introduce a use of the `this` reference that's allowed only as the *first* statement in a constructor's body. Line 14 uses `this` in method-call syntax to invoke the `Time2` constructor that takes three parameters (lines 30–33) with values of 0 for the `hour`, `minute` and `second`. Using the `this` reference as shown here is a popular way to reuse initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body. We use this syn-

tax in four of the five Time2 constructors to make the class easier to maintain and modify. If we need to change how objects of class Time2 are initialized, only the constructor that the class's other constructors call will need to be modified. In fact, even that constructor might not need modification in this example. That constructor simply calls the setTime method to perform the actual initialization, so it's possible that the changes the class might require would be localized to the *set* methods.

Common Programming Error 8.3

It's a compilation error when this *is used in a constructor's body to call another constructor of the same class if that call is not the* first *statement in the constructor. It's also a compilation error when a method attempts to invoke a constructor directly via* this.

Common Programming Error 8.4

A constructor can call methods of the class. Be aware that the instance variables might not yet be initialized, because the constructor is in the process of initializing the object. Using instance variables before they've been initialized properly is a logic error.

Lines 18–21 declare a Time2 constructor with a single int parameter representing the hour, which is passed with 0 for the minute and second to the constructor at lines 30–33. Lines 24–27 declare a Time2 constructor that receives two int parameters representing the hour and minute, which are passed with 0 for the second to the constructor at lines 30–33. Like the no-argument constructor, each of these constructors invokes the constructor at lines 30–33 to minimize code duplication. Lines 30–33 declare the Time2 constructor that receives three int parameters representing the hour, minute and second. This constructor calls setTime to initialize the instance variables.

Lines 36–40 declare a Time2 constructor that receives a reference to another Time2 object. In this case, the values from the Time2 argument are passed to the three-argument constructor at lines 30–33 to initialize the hour, minute and second. Line 39 could have directly accessed the hour, minute and second values of the constructor's argument time with the expressions time.hour, time.minute and time.second—even though hour, minute and second are declared as private variables of class Time2. This is due to a special relationship between objects of the same class. We'll see in a moment why it's preferable to use the *get* methods.

Software Engineering Observation 8.3

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 (including those that are* private*).*

Class Time2*'s* setTime *Method*

Method setTime (lines 45–50) invokes the setHour (lines 53–59), setMinute (lines 62–68) and setSecond (lines 71–77) methods, which ensure that the value supplied for hour is in the range 0 to 23 and the values for minute and second are each in the range 0 to 59. If a value is out of range, each of these methods throws an IllegalArgumentException (lines 58, 67 and 76) indicating which value was out of range.

Notes Regarding Class Time2*'s* set *and* get *Methods and Constructors*

Time2's *set* and *get* methods are called throughout the class. In particular, method setTime calls methods setHour, setMinute and setSecond in lines 47–49, and methods toUni-

versalString and toString call methods getHour, getMinute and getSecond in line 93 and lines 100–101, respectively. In each case, these methods could have accessed the class's private data directly without calling the *set* and *get* methods. 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 since midnight (requiring only 4 bytes of memory). If we made such a change, only the bodies of the methods that access the private data directly would need to change—in particular, the individual *set* and *get* methods for the hour, minute and second. There would be no need to modify the bodies of methods setTime, toUniversalString or toString because they do not access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, each Time2 constructor could include a copy of the appropriate statements from methods setHour, setMinute and setSecond. Doing so may be slightly more efficient, because the extra calls to the constructor and setTime are eliminated. However, *duplicating* statements in multiple methods or constructors makes changing the class's internal data representation more difficult. Having the Time2 constructors call the constructor with three arguments (or even call setTime directly) requires that any changes to the implementation of setTime be made only once. Also, the compiler can optimize programs by removing calls to simple methods and replacing them with the expanded code of their declarations—a technique known as **inlining the code**, which improves program performance.

Software Engineering Observation 8.4

When implementing a method of a class, use the class's set *and* get *methods to access the class's* private *data. This simplifies code maintenance and reduces the likelihood of errors.*

Using Class Time2's Overloaded Constructors

Class Time2Test (Fig. 8.6) invokes the overloaded Time2 constructors (lines 8–12 and 40). Line 8 invokes the no-argument constructor (Fig. 8.5, lines 12–15). Lines 9–13 of the program demonstrate passing arguments to the other Time2 constructors. Line 9 invokes the single-argument constructor that receives an int at lines 18–21 of Fig. 8.5. Line 10 invokes the two-argument constructor at lines 24–27 of Fig. 8.5. Line 11 invokes the three-argument constructor at lines 30–33 of Fig. 8.5. Line 12 invokes the single-argument constructor that takes a Time2 at lines 36–40 of Fig. 8.5. Next, the application displays the String representations of each Time2 object to confirm that it was initialized properly. Line 40 attempts to intialize t6 by creating a new Time2 object and passing three invalid values to the constructor. When the constructor attempts to use the invalid hour value to initialize the object's hour, an IllegalArgumentException occurs. We catch this exception at line 42 and display its error message, which results in the last line of the output.

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test
5  {
```

Fig. 8.6 | Overloaded constructors used to initialize Time2 objects. (Part 1 of 3.)

```
 6     public static void main( String[] args )
 7     {
 8        Time2 t1 = new Time2(); // 00:00:00
 9        Time2 t2 = new Time2( 2 ); // 02:00:00
10        Time2 t3 = new Time2( 21, 34 ); // 21:34:00
11        Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12        Time2 t5 = new Time2( t4 ); // 12:25:42
13
14        System.out.println( "Constructed with:" );
15        System.out.println( "t1: all arguments defaulted" );
16        System.out.printf( "   %s\n", t1.toUniversalString() );
17        System.out.printf( "   %s\n", t1.toString() );
18
19        System.out.println(
20           "t2: hour specified; minute and second defaulted" );
21        System.out.printf( "   %s\n", t2.toUniversalString() );
22        System.out.printf( "   %s\n", t2.toString() );
23
24        System.out.println(
25           "t3: hour and minute specified; second defaulted" );
26        System.out.printf( "   %s\n", t3.toUniversalString() );
27        System.out.printf( "   %s\n", t3.toString() );
28
29        System.out.println( "t4: hour, minute and second specified" );
30        System.out.printf( "   %s\n", t4.toUniversalString() );
31        System.out.printf( "   %s\n", t4.toString() );
32
33        System.out.println( "t5: Time2 object t4 specified" );
34        System.out.printf( "   %s\n", t5.toUniversalString() );
35        System.out.printf( "   %s\n", t5.toString() );
36
37        // attempt to initialize t6 with invalid values
38        try
39        {
40           Time2 t6 = new Time2( 27, 74, 99 ); // invalid values
41        } // end try
42        catch ( IllegalArgumentException e )
43        {
44           System.out.printf( "\nException while initializing t6: %s\n",
45              e.getMessage() );
46        } // end catch
47     } // end main
48  } // end class Time2Test
```

```
Constructed with:
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 2 of 3.)

```
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM

Exception while initializing t6: hour must be 0-23
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 3 of 3.)

8.6 Default and No-Argument Constructors

Every class must have at least one constructor. If you do not provide any in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked. The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, `false` for `boolean` values and `null` for references). In Section 9.4.1, you'll learn that the default constructor performs another task also.

If your class declares constructors, the compiler will *not* create a default constructor. In this case, you must declare a no-argument constructor if default initialization is required. Like a default constructor, a no-argument constructor is invoked with empty parentheses. The `Time2` no-argument constructor (lines 12–15 of Fig. 8.5) explicitly initializes a `Time2` object by passing to the three-argument constructor 0 for each parameter. Since 0 is the default value for `int` instance variables, the no-argument constructor in this example could actually be declared with an empty body. In this case, each instance variable would receive its default value when the no-argument constructor was called. If we omit the no-argument constructor, clients of this class would not be able to create a `Time2` object with the expression `new Time2()`.

Common Programming Error 8.5

A compilation error occurs if a program attempts to initialize an object of a class by passing the wrong number or types of arguments to the class's constructor.

Error-Prevention Tip 8.2

Ensure that you do not *include a return type in a constructor definition. Java allows other methods of the class besides its constructors 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.*

8.7 Notes on *Set* and *Get* Methods

As you know, a class's `private` fields can be manipulated only by its methods. 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.

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`. *Set* methods are also commonly called **mutator methods**, because they typically change an object's state—i.e., modify the values of instance variables. *Get* methods are also commonly called **accessor methods** or **query methods**.

Set *and* Get *Methods vs.* `public` *Data*

It would seem that providing *set* and *get* capabilities is essentially the same as making the instance variables `public`. This is one of the subtleties that makes Java so desirable for software engineering. A `public` instance variable can be read or written by any method that has a reference to an object containing that variable. If an instance variable is declared `private`, a `public` *get* method certainly allows other methods to access it, but the *get* method can *control* how the client can access it. For example, a *get* method might control the format of the data it returns and thus shield the client code from the actual data representation. A `public` *set* method can—and should—carefully scrutinize attempts to modify the variable's value and throw an exception if necessary. For example, an attempt to *set* the day of the month to 37 would be rejected, an attempt to *set* a person's weight to a negative value would be rejected, and so on. Thus, although *set* and *get* methods provide access to `private` data, the access is restricted by the implementation of the methods. This helps promote good software engineering.

Validity Checking in Set *Methods*

The benefits of data integrity do not follow automatically simply because instance variables are declared `private`—you must provide validity checking. Java enables you to design better programs in a convenient manner. A class's *set* methods could return values indicating that attempts were made to assign invalid data to objects of the class. A client of the class could test the return value of a *set* method to determine whether the client's attempt to modify the object was successful and to take appropriate action. Typically, however, *set* methods have `void` return type and use exception handling to indicate attempts to assign invalid data. We discuss exception handling in detail in Chapter 11.

Software Engineering Observation 8.5

When appropriate, provide `public` *methods to change and retrieve the values of* `private` *instance variables. This architecture helps hide the implementation of a class from its clients, which improves program modifiability.*

Error-Prevention Tip 8.3

Using set *and* get *methods helps you create classes that are easier to debug and maintain. If only one method performs a particular task, such as setting the hour in a* `Time2` *object, it's easier to debug and maintain the class. If the* `hour` *is not being set properly, the code that actually modifies instance variable* `hour` *is localized to one method's body—*`setHour`*. Thus, your debugging efforts can be focused on method* `setHour`*.*

Predicate Methods

Another common use for accessor methods is to test whether a condition is true or false—such methods are often called **predicate methods.** An example would be class `ArrayList`'s `isEmpty` method, which returns `true` if the `ArrayList` is empty. A program might test `isEmpty` before attempting to read another item from an `ArrayList`.

8.8 Composition

A class can have references to objects of other classes as members. This is called **composition** and is sometimes referred to as a ***has-a* relationship**. For example, an `AlarmClock` object needs to know the current time *and* the time when it's supposed to sound its alarm, so it's reasonable to include *two* references to `Time` objects in an `AlarmClock` object.

Class `Date`

This composition example contains classes `Date` (Fig. 8.7), `Employee` (Fig. 8.8) and `EmployeeTest` (Fig. 8.9). Class `Date` (Fig. 8.7) declares instance variables month, day and year (lines 6–8) to represent a date. The constructor receives three `int` parameters. Line 17 invokes utility method `checkMonth` (lines 26–32) to validate the month—if the value is out-of-range the method throws an exception. Line 15 assumes that the value for `year` is correct and doesn't validate it. Line 19 invokes utility method `checkDay` (lines 35–48) to validate the `day` based on the current `month` and `year`. Line 38 determines whether the day is correct based on the number of days in the particular `month`. If the day is not correct, lines 42–43 determine whether the `month` is February, the day is `29` and the `year` is a leap year. If the `day` is still invalid, the method throws an exception. Lines 21–22 in the constructor output the `this` reference as a `String`. Since `this` is a reference to the current `Date` object, the object's `toString` method (lines 51–54) is called *implicitly* to obtain the object's `String` representation.

```
1  // Fig. 8.7: Date.java
2  // Date class declaration.
3
4  public class Date
5  {
6     private int month; // 1-12
7     private int day; // 1-31 based on month
8     private int year; // any year
9
10    private static final int[] daysPerMonth = // days in each month
11       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // constructor: call checkMonth to confirm proper value for month;
14    // call checkDay to confirm proper value for day
15    public Date( int theMonth, int theDay, int theYear )
16    {
17       month = checkMonth( theMonth ); // validate month
18       year = theYear; // could validate year
19       day = checkDay( theDay ); // validate day
20
21       System.out.printf(
22          "Date object constructor for date %s\n", this );
23    } // end Date constructor
24
25    // utility method to confirm proper month value
26    private int checkMonth( int testMonth )
27    {
28       if ( testMonth > 0 && testMonth <= 12 ) // validate month
29          return testMonth;
```

Fig. 8.7 | `Date` class declaration. (Part 1 of 2.)

```
30          else // month is invalid
31             throw new IllegalArgumentException( "month must be 1-12" );
32       } // end method checkMonth
33
34       // utility method to confirm proper day value based on month and year
35       private int checkDay( int testDay )
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 && ( year % 400 == 0 ||
43               ( year % 4 == 0 && year % 100 != 0 ) ) )
44             return testDay;
45
46          throw new IllegalArgumentException(
47             "day out-of-range for the specified month and year" );
48       } // end method checkDay
49
50       // return a String of the form month/day/year
51       public String toString()
52       {
53          return String.format( "%d/%d/%d", month, day, year );
54       } // end method toString
55    } // end class Date
```

Fig. 8.7 | `Date` class declaration. (Part 2 of 2.)

Class `Employee`

Class `Employee` (Fig. 8.8) has instance variables `firstName`, `lastName`, `birthDate` and `hireDate`. Members `firstName` and `lastName` (lines 6–7) are references to `String` objects. Members `birthDate` and `hireDate` (lines 8–9) are references to `Date` objects. This demonstrates that a class can have as instance variables references to objects of other classes. The `Employee` constructor (lines 12–19) takes four parameters—`first`, `last`, `dateOfBirth` and `dateOfHire`. The objects referenced by the parameters are assigned to the `Employee` object's instance variables. When class `Employee`'s `toString` method is called, it returns a `String` containing the employee's name and the `String` representations of the two `Date` objects. Each of these `String`s is obtained with an *implicit* call to the `Date` class's `toString` method.

```
1  // Fig. 8.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
```

Fig. 8.8 | `Employee` class with references to other objects. (Part 1 of 2.)

```
10
11    // constructor to initialize name, birth date and hire date
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // end Employee constructor
20
21    // convert Employee to String format
22    public String toString()
23    {
24       return String.format( "%s, %s  Hired: %s  Birthday: %s",
25          lastName, firstName, hireDate, birthDate );
26    } // end method toString
27 } // end class Employee
```

Fig. 8.8 | `Employee` class with references to other objects. (Part 2 of 2.)

Class `EmployeeTest`

Class `EmployeeTest` (Fig. 8.9) creates two `Date` objects (lines 8–9) to represent an `Employee`'s birthday and hire date, respectively. Line 10 creates an `Employee` and initializes its instance variables by passing to the constructor two `String`s (representing the `Employee`'s first and last names) and two `Date` objects (representing the birthday and hire date). Line 12 implicitly invokes the `Employee`'s `toString` method to display the values of its instance variables and demonstrate that the object was initialized properly.

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        Date birth = new Date( 7, 24, 1949 );
9        Date hire = new Date( 3, 12, 1988 );
10       Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12       System.out.println( employee );
13    } // end main
14 } // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. 8.9 | Composition demonstration.

8.9 Enumerations

In Fig. 6.8, we introduced the basic `enum` type, which defines a set of constants represented as unique identifiers. In that program the `enum` constants represented the game's status. In this section we discuss the relationship between `enum` types and classes. Like classes, all `enum` types are reference types. An `enum` type is declared with an **`enum` declaration**, which is a comma-separated list of `enum` constants—the declaration may optionally include other components of traditional classes, such as constructors, fields and methods. Each `enum` declaration declares an `enum` class with the following restrictions:

1. `enum` constants are implicitly `final`, because they declare constants that shouldn't be modified.
2. `enum` constants are implicitly `static`.
3. Any attempt to create an object of an enum type with operator `new` results in a compilation error.

The `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and to control enhanced `for` statements.

Figure 8.10 illustrates how to declare instance variables, a constructor and methods in an `enum` type. The `enum` declaration (lines 5–37) contains two parts—the `enum` constants and the other members of the `enum` type. The first part (lines 8–13) declares six `enum` constants. Each is optionally followed by arguments which are passed to the **`enum` constructor** (lines 20–24). Like the constructors you've seen in classes, an `enum` constructor can specify any number of parameters and can be overloaded. In this example, the `enum` constructor requires two `String` parameters. To properly initialize each `enum` constant, we follow it with parentheses containing two `String` arguments, which are passed to the `enum`'s constructor. The second part (lines 16–36) declares the other members of the `enum` type—two instance variables (lines 16–17), a constructor (lines 20–24) and two methods (lines 27–30 and 33–36).

```
1  // Fig. 8.10: Book.java
2  // Declaring an enum type with constructor and explicit instance fields
3  // and accessors for these fields
4
5  public enum Book
6  {
7     // declare constants of enum type
8     JHTP( "Java How to Program", "2012" ),
9     CHTP( "C How to Program", "2007" ),
10    IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTP( "C++ How to Program", "2012" ),
12    VBHTP( "Visual Basic 2010 How to Program", "2011" ),
13    CSHARPHTP( "Visual C# 2010 How to Program", "2011" );
14
15    // instance fields
16    private final String title; // book title
17    private final String copyrightYear; // copyright year
```

Fig. 8.10 | Declaring an `enum` type with constructor and explicit instance fields and accessors for these fields. (Part 1 of 2.)

```
18
19      // enum constructor
20      Book( String bookTitle, String year )
21      {
22         title = bookTitle;
23         copyrightYear = year;
24      } // end enum Book constructor
25
26      // accessor for field title
27      public String getTitle()
28      {
29         return title;
30      } // end method getTitle
31
32      // accessor for field copyrightYear
33      public String getCopyrightYear()
34      {
35         return copyrightYear;
36      } // end method getCopyrightYear
37   } // end enum Book
```

Fig. 8.10 | Declaring an `enum` type with constructor and explicit instance fields and accessors for these fields. (Part 2 of 2.)

Lines 16–17 declare the instance variables `title` and `copyrightYear`. Each `enum` constant in `Book` is actually an object of type `Book` that has its own copy of instance variables `title` and `copyrightYear`. The constructor (lines 20–24) takes two `String` parameters, one that specifies the book's title and one that specifies its copyright year. Lines 22–23 assign these parameters to the instance variables. Lines 27–36 declare two methods, which return the book title and copyright year, respectively.

Figure 8.11 tests the `enum` type `Book` and illustrates how to iterate through a range of `enum` constants. For every `enum`, the compiler generates the `static` method **`values`** (called in line 12) that returns an array of the `enum`'s constants in the order they were declared. Lines 12–14 use the enhanced `for` statement to display all the constants declared in the `enum` `Book`. Line 14 invokes the `enum` `Book`'s `getTitle` and `getCopyrightYear` methods to get the title and copyright year associated with the constant. When an `enum` constant is converted to a `String` (e.g., `book` in line 13), the constant's identifier is used as the `String` representation (e.g., `JHTP` for the first `enum` constant).

```
1  // Fig. 8.11: EnumTest.java
2  // Testing enum type Book.
3  import java.util.EnumSet;
4
5  public class EnumTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "All books:\n" );
```

Fig. 8.11 | Testing an `enum` type. (Part 1 of 2.)

```
10
11         // print all books in enum Book
12         for ( Book book : Book.values() )
13            System.out.printf( "%-10s%-45s%s\n", book,
14                book.getTitle(), book.getCopyrightYear() );
15
16         System.out.println( "\nDisplay a range of enum constants:\n" );
17
18         // print first four books
19         for ( Book book : EnumSet.range( Book.JHTP, Book.CPPHTP ) )
20            System.out.printf( "%-10s%-45s%s\n", book,
21                book.getTitle(), book.getCopyrightYear() );
22      } // end main
23   } // end class EnumTest
```

```
All books:

JHTP       Java How to Program                           2012
CHTP       C How to Program                              2007
IW3HTP     Internet & World Wide Web How to Program      2008
CPPHTP     C++ How to Program                            2012
VBHTP      Visual Basic 2010 How to Program              2011
CSHARPHTP  Visual C# 2010 How to Program                 2011

Display a range of enum constants:

JHTP       Java How to Program                           2012
CHTP       C How to Program                              2007
IW3HTP     Internet & World Wide Web How to Program      2008
CPPHTP     C++ How to Program                            2012
```

Fig. 8.11 | Testing an `enum` type. (Part 2 of 2.)

Lines 19–21 use the `static` method **`range`** of class **`EnumSet`** (declared in package `java.util`) to display a range of the `enum` `Book`'s constants. Method `range` takes two parameters—the first and the last `enum` constants in the range—and returns an `EnumSet` that contains all the constants between these two constants, inclusive. For example, the expression `EnumSet.range( Book.JHTP, Book.CPPHTP )` returns an `EnumSet` containing `Book.JHTP`, `Book.CHTP`, `Book.IW3HTP` and `Book.CPPHTP`. The enhanced `for` statement can be used with an `EnumSet` just as it can with an array, so lines 12–14 use it to display the title and copyright year of every book in the `EnumSet`. Class `EnumSet` provides several other `static` methods for creating sets of `enum` constants from the same `enum` type.

Common Programming Error 8.6

In an `enum` declaration, it's a syntax error to declare `enum` constants after the `enum` type's constructors, fields and methods.

8.10 Garbage Collection and Method `finalize`

Every class in Java has the methods of class `Object` (package `java.lang`), one of which is the `finalize` method. This method is rarely used because it can cause performance problems and there's some uncertainty as to whether it will get called. Nevertheless, because

`finalize` is part of every class, we discuss it here to help you understand its intended purpose. The complete details of the `finalize` method are beyond the scope of this book, and most programmers should not use it—you'll soon see why. You'll learn more about class `Object` in Chapter 9.

Every object uses system resources, such as memory. We need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur that would prevent them from being reused by your program or possibly by other programs. The JVM performs automatic **garbage collection** to reclaim the memory occupied by objects that are no longer used. When there are no more references to an object, the object is eligible to be collected. This typically occurs when the JVM executes its **garbage collector**. So, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways. Other types of resource leaks can occur. For example, an application may open a file on disk to modify its contents. If it does not close the file, the application must terminate before any other application can use it.

The **`finalize` method** is called by the garbage collector to perform **termination housekeeping** on an object just before the garbage collector reclaims the object's memory. Method `finalize` does not take parameters and has return type `void`. A problem with method `finalize` is that the garbage collector is not guaranteed to execute at a specified time. In fact, the garbage collector may never execute before a program terminates. Thus, it's unclear whether, or when, method `finalize` will be called. For this reason, most programmers should avoid method `finalize`.

Software Engineering Observation 8.6

A class that uses system resources, such as files on disk, should provide a method that programmers can call to release resources when they're no longer needed in a program. Many Java API classes provide `close` *or* `dispose` *methods for this purpose. For example, class* `Scanner` *has a* `close` *method. We discuss new Java SE 7 features related to this in Section 11.13.*

8.11 `static` Class Members

Every object 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` field**—called a **class variable**—is used in such cases. A `static` variable represents **classwide information**—all objects of the class share the *same* piece of data. The declaration of a `static` variable begins with the keyword `static`.

Let's motivate `static` data with an example. Suppose that 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 at least four other `Martians` are present. If fewer than five `Martians` are present, each of them becomes cowardly. Thus, each `Martian` needs to know the `martianCount`. We could endow class `Martian` with `martianCount` as an instance variable. If we do this, then every `Martian` will have *a separate copy* of the instance variable, and every time we create a new `Martian`, we'll have to update the instance variable `martianCount` in every `Martian` object. This wastes space with the redundant copies, wastes time in updating the separate copies and is error prone. Instead, we declare `martianCount` to be `static`, making `martianCount` classwide data. Every `Mar-`

tian can see the martianCount as if it were an instance variable of class Martian, but only one copy of the static martianCount is maintained. This saves space. We save time by having the Martian constructor increment the static martianCount—there's only one copy, so we do not have to increment separate copies for each Martian object.

Software Engineering Observation 8.7

Use a static variable when all objects of a class must use the same copy of the variable.

Static variables have class scope. We can access a class's public static members through a reference to any object of the class, or by qualifying the member name with the class name and a dot (.), as in Math.random(). A class's private static class members can be accessed by client code only through methods of the class. Actually, *static class members exist even when no objects of the class exist*—they're available as soon as the class is loaded into memory at execution time. To access a public static member when no objects of the class exist (and even when they do), prefix the class name and a dot (.) to the static member, as in Math.PI. To access a private static member when no objects of the class exist, provide a public static method and call it by qualifying its name with the class name and a dot.

Software Engineering Observation 8.8

Static class variables and methods exist, and can be used, even if no objects of that class have been instantiated.

A static method cannot access non-static class members, because a static method can be called even when no objects of the class have been instantiated. For the same reason, the this reference cannot be used in a static method. The this reference must refer to a specific object of the class, and when a static method is called, there might not be any objects of its class in memory.

Common Programming Error 8.7

A compilation error occurs if a static method calls an instance (non-static) method in the same class by using only the method name. Similarly, a compilation error occurs if a static method attempts to access an instance variable in the same class by using only the variable name.

Common Programming Error 8.8

Referring to this in a static method is a compilation error.

Tracking the Number of Employee Objects That Have Been Created

Our next program declares two classes—Employee (Fig. 8.12) and EmployeeTest (Fig. 8.13). Class Employee declares private static variable count (Fig. 8.12, line 9) and public static method getCount (lines 36–39). The static variable count is initialized to zero in line 9. If a static variable is not initialized, the compiler assigns it a default value—in this case 0, the default value for type int. Variable count maintains a count of the number of objects of class Employee that have been created so far.

When Employee objects exist, variable count can be used in any method of an Employee object—this example increments count in the constructor (line 18). The public

```
1  // Fig. 8.12: Employee.java
2  // Static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // number of Employees created
10
11    // initialize Employee, add 1 to static count and
12    // output String indicating that constructor was called
13    public Employee( String first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       ++count;  // increment static count of employees
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // end Employee constructor
22
23    // get first name
24    public String getFirstName()
25    {
26       return firstName;
27    } // end method getFirstName
28
29    // get last name
30    public String getLastName()
31    {
32       return lastName;
33    } // end method getLastName
34
35    // static method to get static count value
36    public static int getCount()
37    {
38       return count;
39    } // end method getCount
40 } // end class Employee
```

Fig. 8.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory.

`static` method `getCount` (lines 36–39) returns the number of `Employee` objects that have been created so far. When no objects of class `Employee` exist, client code can access variable `count` by calling method `getCount` via the class name, as in `Employee.getCount()`. When objects exist, method `getCount` can also be called via any reference to an `Employee` object.

Good Programming Practice 8.1

Invoke every `static` method by using the class name and a dot (.) to emphasize that the method being called is a `static` method.

`EmployeeTest` method `main` (Fig. 8.13) instantiates two `Employee` objects (lines 13–14). When each `Employee` object's constructor is invoked, lines 15–16 of Fig. 8.12 assign the `Employee`'s first name and last name to instance variables `firstName` and `lastName`. These two statements do *not* make copies of the original `String` arguments. Actually, `String` objects in Java are **immutable**—they cannot be modified after they're created. Therefore, it's safe to have many references to one `String` object. This is not normally the case for objects of most other classes in Java. If `String` objects are immutable, you might wonder why we're able to use operators + and += to concatenate `String` objects. String-concatenation operations actually result in a *new* `String` object containing the concatenated values. The original `String` objects are not modified.

When `main` has finished using the two `Employee` objects, the references `e1` and `e2` are set to `null` at lines 31–32. At this point, references `e1` and `e2` no longer refer to the objects that were instantiated in lines 13–14. The objects become "eligible for garbage collection" because there are no more references to them in the program.

```
1  // Fig. 8.13: EmployeeTest.java
2  // static member demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // show that count is 0 before creating Employees
9        System.out.printf( "Employees before instantiation: %d\n",
10          Employee.getCount() );
11
12       // create two Employees; count should be 2
13       Employee e1 = new Employee( "Susan", "Baker" );
14       Employee e2 = new Employee( "Bob", "Blue" );
15
16       // show that count is 2 after creating two Employees
17       System.out.println( "\nEmployees after instantiation: " );
18       System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19       System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20       System.out.printf( "via Employee.getCount(): %d\n",
21          Employee.getCount() );
22
23       // get names of Employees
24       System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25          e1.getFirstName(), e1.getLastName(),
26          e2.getFirstName(), e2.getLastName() );
27
28       // in this example, there is only one reference to each Employee,
29       // so the following two statements indicate that these objects
30       // are eligible for garbage collection
31       e1 = null;
32       e2 = null;
33    } // end main
34 } // end class EmployeeTest
```

Fig. 8.13 | `static` member demonstration. (Part 1 of 2.)

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Fig. 8.13 | `static` member demonstration. (Part 2 of 2.)

Eventually, the garbage collector might reclaim the memory for these objects (or the operating system will reclaim the memory when the program terminates). The JVM does not guarantee when, or even whether, the garbage collector will execute. When it does, it's possible that no objects or only a subset of the eligible objects will be collected.

8.12 `static` Import

In Section 6.3, you learned about the `static` fields and methods of class `Math`. We invoked class `Math`'s `static` fields and methods by preceding each with the class name `Math` and a dot (`.`). A **`static` import** declaration enables you to import the `static` members of a class or interface so you can access them via their unqualified names in your class—the class name and a dot (`.`) are not required to use an imported `static` member.

A `static` import declaration has two forms—one that imports a particular `static` member (which is known as **single `static` import**) and one that imports *all* `static` members of a class (known as **`static` import on demand**). The following syntax imports a particular `static` member:

```
import static packageName.ClassName.staticMemberName;
```

where *packageName* is the package of the class (e.g., `java.lang`), *ClassName* is the name of the class (e.g., `Math`) and *staticMemberName* is the name of the `static` field or method (e.g., `PI` or `abs`). The following syntax imports all `static` members of a class:

```
import static packageName.ClassName.*;
```

The asterisk (`*`) indicates that *all* `static` members of the specified class should be available for use in the file. `static` import declarations import only `static` class members. Regular `import` statements should be used to specify the classes used in a program.

Figure 8.14 demonstrates a `static` import. Line 3 is a `static` import declaration, which imports all `static` fields and methods of class `Math` from package `java.lang`. Lines 9–12 access the `Math` class's `static` fields `E` (line 11) and `PI` (line 12) and the `static` methods `sqrt` (line 9) and `ceil` (line 10) without preceding the field names or method names with class name `Math` and a dot.

Common Programming Error 8.9

A compilation error occurs if a program attempts to import two or more classes' `static` methods that have the same signature or `static` fields that have the same name.

```
1  // Fig. 8.14: StaticImportTest.java
2  // Static import of Math class methods.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "E = %f\n", E );
12       System.out.printf( "PI = %f\n", PI );
13    } // end main
14 } // end class StaticImportTest
```

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Fig. 8.14 | Static import of `Math` class methods.

8.13 final Instance Variables

The **principle of least privilege** is fundamental to good software engineering. In the context of an application, it states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more. This makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.

Let's see how this principle applies to instance variables. Some of them need to be modifiable and some do not. You can use the keyword `final` to specify that a variable is not modifiable (i.e., it's a constant) and that any attempt to modify it is an error. For example,

```
private final int INCREMENT;
```

declares a `final` (constant) instance variable `INCREMENT` of type `int`. Such variables can be initialized when they're declared. If they are not, they *must* be initialized in every constructor of the class. Initializing constants in constructors enables each object of the class to have a different value for the constant. If a `final` variable is not initialized in its declaration or in every constructor, a compilation error occurs.

Software Engineering Observation 8.9

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 prevent modification.

Common Programming Error 8.10

Attempting to modify a `final` instance variable after it's initialized is a compilation error.

Error-Prevention Tip 8.4

Attempts to modify a `final` *instance variable are caught at compilation time rather than causing execution-time errors. It's always preferable to get bugs out at compilation time, if possible, rather than allow them to slip through to execution time (where experience has found that repair is often many times more expensive).*

Software Engineering Observation 8.10

A `final` *field should also be declared* `static` *if it's initialized in its declaration to a value that's the same for all objects of the class. After this initialization, its value can never change. Therefore, we don't need a separate copy of the field for every object of the class. Making the field* `static` *enables all objects of the class to share the* `final` *field.*

8.14 `Time` Class Case Study: Creating Packages

We've seen in almost every example in the text that classes from preexisting libraries, such as the Java API, can be imported into a Java program. Each class in the Java API belongs to a package that contains a group of related classes. These packages are defined once, but can be imported into many programs. As applications become more complex, packages help you manage the complexity of application components. Packages also facilitate software reuse by enabling programs to *import* classes from other packages (as we've done in most examples), rather than *copying* the classes into each program that uses them. Another benefit of packages is that they provide a convention for unique class names, which helps prevent class-name conflicts (discussed later in this section). This section introduces how to create your own packages.

Steps for Declaring a Reusable Class

Before a class can be imported into multiple applications, it must be placed in a package to make it reusable. Figure 8.15 shows how to specify the package in which a class should be placed. Figure 8.16 shows how to import our packaged class so that it can be used in an application. The steps for creating a reusable class are:

1. Declare a `public` class. If the class is not `public`, it can be used only by other classes in the same package.
2. Choose a unique package name and add a **package declaration** to the source-code file for the reusable class declaration. In each Java source-code file there can be only one `package` declaration, and it must precede all other declarations and statements. Comments are not statements, so comments can be placed before a `package` statement in a file. [*Note:* If no `package` statement is provided, the class is placed in the so-called default package and is accessible only to other classes in the default package that are located in the same directory. All prior programs in this book having two or more classes have used this default package.]
3. Compile the class so that it's placed in the appropriate package directory.
4. Import the reusable class into a program and use the class.

We'll now discuss each of these steps in detail.

Steps 1 and 2: Creating a `public` *Class and Adding the* `package` *Statement*

For *Step 1*, we modify the `public` class `Time1` declared in Fig. 8.1. The new version is shown in Fig. 8.15. No modifications have been made to the implementation of the class, so we'll not discuss its implementation details again here.

For *Step 2*, we add a `package` declaration (line 3) that declares a `package` named `com.deitel.jhtp.ch08`. Placing a `package` declaration at the beginning of a Java source file indicates that the class declared in the file is part of the specified package. Only `package` declarations, `import` declarations and comments can appear outside the braces of a class declaration. A Java source-code file must have the following order:

```
1  // Fig. 8.15: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3  package com.deitel.jhtp.ch08;
4
5  public class Time1
6  {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // set a new time value using universal time; throw an
12    // exception if the hour, minute or second is invalid
13    public void setTime( int h, int m, int s )
14    {
15       // validate hour, minute and second
16       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
17          ( s >= 0 && s < 60 ) )
18       {
19          hour = h;
20          minute = m;
21          second = s;
22       } // end if
23       else
24          throw new IllegalArgumentException(
25             "hour, minute and/or second was out of range" );
26    } // end method setTime
27
28    // convert to String in universal-time format (HH:MM:SS)
29    public String toUniversalString()
30    {
31       return String.format( "%02d:%02d:%02d", hour, minute, second );
32    } // end method toUniversalString
33
34    // convert to String in standard-time format (H:MM:SS AM or PM)
35    public String toString()
36    {
37       return String.format( "%d:%02d:%02d %s",
38          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
39          minute, second, ( hour < 12 ? "AM" : "PM" ) );
40    } // end method toString
41 } // end class Time1
```

Fig. 8.15 | Packaging class `Time1` for reuse.

1. a `package` declaration (if any),
2. `import` declarations (if any), then
3. class declarations.

Only one of the class declarations in a particular file can be `public`. Other classes in the file are placed in the package and can be used only by the other classes in the package. Non-`public` classes are in a package to support the reusable classes in the package.

To provide unique package names, start each one with your Internet domain name in reverse order. For example, our domain name is `deitel.com`, so our package names begin with `com.deitel`. For the domain name *yourcollege*`.edu`, the package name should begin with `edu.`*yourcollege*. After the domain name is reversed, you can choose any other names you want for your package. If you're 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 `jhtp` as the next name in our package name to indicate that this class is from *Java How to Program*. The last name in our package name specifies that this package is for Chapter 8 (`ch08`).

Step 3: Compiling the Packaged Class

Step 3 is to compile the class so that it's stored in the appropriate package. When a Java file containing a `package` declaration is compiled, the resulting class file is placed in the directory specified by the declaration. The `package` declaration in Fig. 8.15 indicates that class `Time1` should be placed in the directory

```
com
   deitel
      jhtp
         ch08
```

The names in the `package` declaration specify the exact location of the package's classes.

When compiling a class in a package, the `javac` command-line option **`-d`** causes the `javac` compiler to create appropriate directories based on the class's `package` declaration. The option also specifies where the directories should be stored. For example, in a command window, we used the compilation command

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

to specify that the first directory in our package name should be placed in the current directory. The period (`.`) after `-d` in the preceding command represents the current directory on the Windows, UNIX, Linux and Mac OS X operating systems (and several others as well). After execution of the compilation command, the current directory contains a directory called `com`, `com` contains a directory called `deitel`, `deitel` contains a directory called `jhtp` and `jhtp` contains a directory called `ch08`. In the `ch08` directory, you can find the file `Time1.class`. [*Note:* If you do not use the `-d` option, then you must copy or move the class file to the appropriate package directory after compiling it.]

The `package` name is part of the **fully qualified class name**, so the name of class `Time1` is actually `com.deitel.jhtp.ch08.Time1`. You can use this fully qualified name in your programs, or you can `import` the class and use its **simple name** (the class name by itself—`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 **name conflict** (also called a **name collision**).

Step 4: Importing the Reusable Class

Once it's compiled and stored in its package, the class can be imported into programs (*Step 4*). In the `Time1PackageTest` application of Fig. 8.16, line 3 specifies that class `Time1` should be imported for use in class `Time1PackageTest`. This class is in the default package because its `.java` file does not contain a `package` declaration. Since the two classes are in different packages, the `import` at line 3 is required so that class `Time1PackageTest` can use class `Time1`.

```
1  // Fig. 8.16: Time1PackageTest.java
2  // Time1 object used in an application.
3  import com.deitel.jhtp.ch08.Time1; // import class Time1
4
5  public class Time1PackageTest
6  {
7     public static void main( String[] args )
8     {
9        // create and initialize a Time1 object
10       Time1 time = new Time1(); // invokes Time1 constructor
11
12       // output string representations of the time
13       System.out.print( "The initial universal time is: " );
14       System.out.println( time.toUniversalString() );
15       System.out.print( "The initial standard time is: " );
16       System.out.println( time.toString() );
17       System.out.println(); // output a blank line
18
19       // change time and output updated time
20       time.setTime( 13, 27, 6 );
21       System.out.print( "Universal time after setTime is: " );
22       System.out.println( time.toUniversalString() );
23       System.out.print( "Standard time after setTime is: " );
24       System.out.println( time.toString() );
25       System.out.println(); // output a blank line
26
27       // attempt to set time with invalid values
28       try
29       {
30          time.setTime( 99, 99, 99 ); // all values out of range
31       } // end try
32       catch ( IllegalArgumentException e )
33       {
34          System.out.printf( "Exception: %s\n\n", e.getMessage() );
35       } // end catch
36
37       // display time after attempt to set invalid values
38       System.out.println( "After attempting invalid settings:" );
39       System.out.print( "Universal time: " );
40       System.out.println( time.toUniversalString() );
41       System.out.print( "Standard time: " );
42       System.out.println( time.toString() );
43    } // end main
44 } // end class Time1PackageTest
```

Fig. 8.16 | `Time1` object used in an application. (Part 1 of 2.)

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.16 | `Time1` object used in an application. (Part 2 of 2.)

Line 3 is known as a **single-type-import declaration**—that is, the `import` declaration specifies one class to import. When your program uses multiple classes from the same package, you can import those classes with a single `import` declaration. For example, the `import` declaration

```
import java.util.*; // import classes from package java.util
```

uses an asterisk (*) at its end to inform the compiler that all `public` classes from the `java.util` package are available for use in the program. This is known as a **type-import-on-demand declaration**. Only the classes from package `java.util` that are used in the program are loaded by the JVM. The preceding `import` allows you to use the simple name of any class from the `java.util` package in the program. Throughout this book, we use single-type-import declarations for clarity.

Common Programming Error 8.11

Using the `import` declaration `import java.*;` causes a compilation error. You must specify the exact name of the package from which you want to import classes.

Specifying the Classpath During Compilation

When compiling `Time1PackageTest`, `javac` must locate the `.class` file for `Time1` to ensure that class `Time1PackageTest` uses class `Time1` correctly. The compiler uses a special object called a **class loader** to locate the classes it needs. The class loader begins by searching the standard Java classes that are bundled with the JDK. Then it searches for **optional packages**. Java provides an **extension mechanism** that enables new (optional) packages to be added to Java for development and execution purposes. If the class is not found in the standard Java classes or in the extension classes, the class loader searches the **classpath**, which contains a list of locations in which classes are stored. The classpath consists of a list of directories or **archive files**, each separated by a **directory separator**—a semicolon (;) on Windows or a colon (:) on UNIX/Linux/Mac OS X. Archive files are individual files that contain directories of other files, typically in a compressed format. For example, the standard classes used by your programs are contained in the archive file `rt.jar`, which is installed with the JDK. Archive files normally end with the `.jar` or `.zip` file-name extensions. The directories and archive files specified in the classpath contain the classes you wish to make available to the Java compiler and the JVM.

By default, the classpath consists only of the current directory. However, the classpath can be modified by

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).

For more information on the classpath, visit download.oracle.com/javase/6/docs/technotes/tools/index.html#general. The section entitled "General Information" contains information on setting the classpath for UNIX/Linux and Windows.

Common Programming Error 8.12

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (.) in the classpath to specify the current directory.

Software Engineering Observation 8.11

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

Error-Prevention Tip 8.5

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

For Figs. 8.15–8.16, we didn't specify an explicit classpath. Thus, to locate the classes in the com.deitel.jhtp.ch08 package from this example, the class loader looks in the current directory for the first name in the package—com—then navigates the directory structure. Directory com contains the subdirectory deitel, deitel contains the subdirectory jhtp, and jhtp contains subdirectory ch08. In the ch08 directory is the file Time1.class, which is loaded by the class loader to ensure that the class is used properly in our program.

Specifying the Classpath When Executing an Application

When you execute an application, the JVM must be able to locate the .class files of the classes used in that application. Like the compiler, the java command uses a class loader that searches the standard classes and extension classes first, then searches the classpath (the current directory by default). The classpath can be specified explicitly by using either of the techniques discussed for the compiler. As with the compiler, it's better to specify an individual program's classpath via command-line JVM options. You can specify the classpath in the java command via the **-classpath** or **-cp** command-line options, followed by a list of directories or archive files separated by semicolons (;) on Microsoft Windows or by colons (:) on UNIX/Linux/Mac OS X. Again, if classes must be loaded from the current directory, be sure to include a dot (.) in the classpath to specify the current directory.

8.15 Package Access

If no access modifier (public, protected or private—we discuss protected in Chapter 9) is specified for a method or variable when it's declared in a class, the method or variable is considered to have **package access**. In a program that consists of one class

declaration, this has no specific effect. However, if a program uses multiple classes from the same package (i.e., a group of related classes), these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of `static` members through the class name. Package access is rarely used.

The application in Fig. 8.17 demonstrates package access. The application contains two classes in one source-code file—the `PackageDataTest` application class (lines 5–21) and the `PackageData` class (lines 24–41). When you compile this program, the compiler produces two separate .class files—`PackageDataTest.class` and `PackageData.class`. The compiler places the two `.class` files in the same directory, so the classes are considered to be part of the same package. Consequently, class `PackageDataTest` is allowed to modify the package-access data of `PackageData` objects. You can also place class `PackageData` (lines 24–41) in a separate source-code file. As long as both classes are compiled in the same directory on disk, the package-access relationship will still work.

```
1  // Fig. 8.17: PackageDataTest.java
2  // Package-access members of a class are accessible by other classes
3  // in the same package.
4
5  public class PackageDataTest
6  {
7     public static void main( String[] args )
8     {
9        PackageData packageData = new PackageData();
10
11       // output String representation of packageData
12       System.out.printf( "After instantiation:\n%s\n", packageData );
13
14       // change package access data in packageData object
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // output String representation of packageData
19       System.out.printf( "\nAfter changing values:\n%s\n", packageData );
20    } // end main
21 } // end class PackageDataTest
22
23 // class with package access instance variables
24 class PackageData
25 {
26    int number; // package-access instance variable
27    String string; // package-access instance variable
28
29    // constructor
30    public PackageData()
31    {
32       number = 0;
33       string = "Hello";
34    } // end PackageData constructor
35
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 1 of 2.)

```
36      // return PackageData object String representation
37      public String toString()
38      {
39         return String.format( "number: %d; string: %s", number, string );
40      } // end method toString
41   } // end class PackageData
```

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 2 of 2.)

In the `PackageData` class declaration, lines 26–27 declare the instance variables `number` and `string` with no access modifiers—therefore, these are package-access instance variables. The `PackageDataTest` application's `main` method creates an instance of the `PackageData` class (line 9) to demonstrate the ability to modify the `PackageData` instance variables directly (as shown in lines 15–16). The results of the modification can be seen in the output window.

8.16 (Optional) GUI and Graphics Case Study: Using Objects with Graphics

Most of the graphics you've seen to this point did not vary with each program execution. Exercise 6.2 in Section 6.13 asked you to create a program that generated shapes and colors at random. In that exercise, the drawing changed every time the system called `paintComponent` to redraw the panel. To create a more consistent drawing that remains the same each time it's drawn, we must store information about the displayed shapes so that we can reproduce them each time the system calls `paintComponent`. To do this, we'll create a set of shape classes that store information about each shape. We'll make these classes "smart" by allowing objects of these classes to draw themselves by using a `Graphics` object.

Class `MyLine`

Figure 8.18 declares class `MyLine`, which has all these capabilities. Class `MyLine` imports `Color` and `Graphics` (lines 3–4). Lines 8–11 declare instance variables for the coordinates needed to draw a line, and line 12 declares the instance variable that stores the color of the line. The constructor at lines 15–22 takes five parameters, one for each instance variable that it initializes. Method `draw` at lines 25–29 requires a `Graphics` object and uses it to draw the line in the proper color and at the proper coordinates.

```
1  // Fig. 8.18: MyLine.java
2  // MyLine class represents a line.
3  import java.awt.Color;
4  import java.awt.Graphics;
```

Fig. 8.18 | `MyLine` class represents a line. (Part 1 of 2.)

```
5
6  public class MyLine
7  {
8     private int x1; // x-coordinate of first endpoint
9     private int y1; // y-coordinate of first endpoint
10    private int x2; // x-coordinate of second endpoint
11    private int y2; // y-coordinate of second endpoint
12    private Color myColor; // color of this shape
13
14    // constructor with input values
15    public MyLine( int x1, int y1, int x2, int y2, Color color )
16    {
17       this.x1 = x1; // set x-coordinate of first endpoint
18       this.y1 = y1; // set y-coordinate of first endpoint
19       this.x2 = x2; // set x-coordinate of second endpoint
20       this.y2 = y2; // set y-coordinate of second endpoint
21       myColor = color; // set the color
22    } // end MyLine constructor
23
24    // Draw the line in the specified color
25    public void draw( Graphics g )
26    {
27       g.setColor( myColor );
28       g.drawLine( x1, y1, x2, y2 );
29    } // end method draw
30 } // end class MyLine
```

Fig. 8.18 | `MyLine` class represents a line. (Part 2 of 2.)

Class `DrawPanel`

In Fig. 8.19, we declare class `DrawPanel`, which will generate random objects of class `MyLine`. Line 12 declares a `MyLine` array to store the lines to draw. Inside the constructor (lines 15–37), line 17 sets the background color to `Color.WHITE`. Line 19 creates the array with a random length between 5 and 9. The loop at lines 22–36 creates a new `MyLine` for every element in the array. Lines 25–28 generate random coordinates for each line's endpoints, and lines 31–32 generate a random color for the line. Line 35 creates a new `MyLine` object with the randomly generated values and stores it in the array.

Method `paintComponent` iterates through the `MyLine` objects in array `lines` using an enhanced `for` statement (lines 45–46). Each iteration calls the `draw` method of the current `MyLine` object and passes it the `Graphics` object for drawing on the panel.

```
1  // Fig. 8.19: DrawPanel.java
2  // Program that uses class MyLine
3  // to draw random lines.
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import java.util.Random;
7  import javax.swing.JPanel;
8
```

Fig. 8.19 | Creating random `MyLine` objects. (Part 1 of 2.)

```
 9  public class DrawPanel extends JPanel
10  {
11     private Random randomNumbers = new Random();
12     private MyLine[] lines; // array of lines
13
14     // constructor, creates a panel with random shapes
15     public DrawPanel()
16     {
17        setBackground( Color.WHITE );
18
19        lines = new MyLine[ 5 + randomNumbers.nextInt( 5 ) ];
20
21        // create lines
22        for ( int count = 0; count < lines.length; count++ )
23        {
24           // generate random coordinates
25           int x1 = randomNumbers.nextInt( 300 );
26           int y1 = randomNumbers.nextInt( 300 );
27           int x2 = randomNumbers.nextInt( 300 );
28           int y2 = randomNumbers.nextInt( 300 );
29
30           // generate a random color
31           Color color = new Color( randomNumbers.nextInt( 256 ),
32              randomNumbers.nextInt( 256 ), randomNumbers.nextInt( 256 ) );
33
34           // add the line to the list of lines to be displayed
35           lines[ count ] = new MyLine( x1, y1, x2, y2, color );
36        } // end for
37     } // end DrawPanel constructor
38
39     // for each shape array, draw the individual shapes
40     public void paintComponent( Graphics g )
41     {
42        super.paintComponent( g );
43
44        // draw the lines
45        for ( MyLine line : lines )
46           line.draw( g );
47     } // end method paintComponent
48  } // end class DrawPanel
```

Fig. 8.19 | Creating random `MyLine` objects. (Part 2 of 2.)

Class `TestDraw`

Class `TestDraw` in Fig. 8.20 sets up a new window to display our drawing. Since we're setting the coordinates for the lines only once in the constructor, the drawing does not change if `paintComponent` is called to refresh the drawing on the screen.

```
1  // Fig. 8.20: TestDraw.java
2  // Creating a JFrame to display a DrawPanel.
3  import javax.swing.JFrame;
```

Fig. 8.20 | Creating a `JFrame` to display a `DrawPanel`. (Part 1 of 2.)

```
4
5  public class TestDraw
6  {
7     public static void main( String[] args )
8     {
9        DrawPanel panel = new DrawPanel();
10       JFrame application = new JFrame();
11
12       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       application.add( panel );
14       application.setSize( 300, 300 );
15       application.setVisible( true );
16    } // end main
17 } // end class TestDraw
```

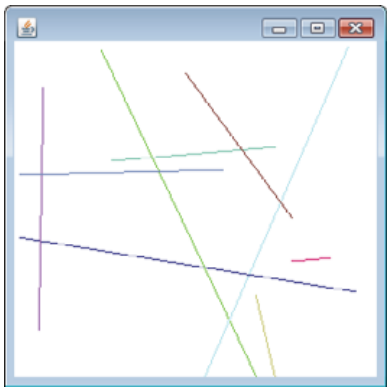


Fig. 8.20 | Creating a `JFrame` to display a `DrawPanel`. (Part 2 of 2.)

GUI and Graphics Case Study Exercise

8.1 Extend the program in Figs. 8.18–8.20 to randomly draw rectangles and ovals. Create classes `MyRectangle` and `MyOval`. Both of these classes should include *x1*, *y1*, *x2*, *y2* coordinates, a color and a `boolean` flag to determine whether the shape is filled. Declare a constructor in each class with arguments for initializing all the instance variables. To help draw rectangles and ovals, each class should provide methods `getUpperLeftX`, `getUpperLeftY`, `getWidth` and `getHeight` that calculate the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height, respectively. The upper-left *x*-coordinate is the smaller of the two *x*-coordinate values, the upper-left *y*-coordinate is the smaller of the two *y*-coordinate values, the width is the absolute value of the difference between the two *x*-coordinate values, and the height is the absolute value of the difference between the two *y*-coordinate values.

Class `DrawPanel`, which extends `JPanel` and handles the creation of the shapes, should declare three arrays, one for each shape type. The length of each array should be a random number between 1 and 5. The constructor of class `DrawPanel` will fill each array with shapes of random position, size, color and fill.

In addition, modify all three shape classes to include the following:

a) A constructor with no arguments that sets the shape's coordinates to `0`, the color of the shape to `Color.BLACK`, and the filled property to `false` (`MyRectangle` and `MyOval` only).

b) *Set* methods for the instance variables in each class. The methods that set a coordinate value should verify that the argument is greater than or equal to zero before setting the coordinate—if it's not, they should set the coordinate to zero. The constructor should call the *set* methods rather than initialize the local variables directly.

c) *Get* methods for the instance variables in each class. Method `draw` should reference the coordinates by the *get* methods rather than access them directly.

8.17 Wrap-Up

In this chapter, we presented additional class concepts. The `Time` class case study presented a complete class declaration consisting of `private` data, overloaded `public` constructors for initialization flexibility, *set* and *get* methods for manipulating the class's data, and methods that returned `String` representations of a `Time` object in two different formats. You also learned that every class can declare a `toString` method that returns a `String` representation of an object of the class and that method `toString` can be called implicitly whenever an object of a class appears in the code where a `String` is expected.

You learned that the `this` reference is used implicitly in a class's non-`static` methods to access the class's instance variables and other non-`static` methods. You also saw explicit uses of the `this` reference to access the class's members (including shadowed fields) and how to use keyword `this` in a constructor to call another constructor of the class.

We discussed the differences between default constructors provided by the compiler and no-argument constructors provided by the programmer. You learned that a class can have references to objects of other classes as members—a concept known as composition. You saw the `enum` class type and learned how it can be used to create a set of constants for use in a program. You learned about Java's garbage-collection capability and how it (unpredictably) reclaims the memory of objects that are no longer used. The chapter explained the motivation for `static` fields in a class and demonstrated how to declare and use `static` fields and methods in your own classes. You also learned how to declare and initialize `final` variables.

You learned how to package your own classes for reuse and how to import those classes into an application. Finally, you learned that fields declared without an access modifier are given package access by default. You saw the relationship between classes in the same package that allows each class in a package to access the package-access members of other classes in the package.

In the next chapter, you'll learn about an important aspect of object-oriented programming in Java—inheritance. You'll see that all classes in Java are related directly or indirectly to the class called `Object`. You'll also begin to understand how the relationships between classes enable you to build more powerful applications.

Summary

Section 8.2 `Time` Class Case Study

- The `public` methods of a class are also known as the class's `public` services or `public` interface (p. 312). They present to the class's clients a view of the services the class provides.
- A class's `private` members are not accessible to its clients.
- `String` class `static` method `format` (p. 314) is similar to method `System.out.printf` except that `format` returns a formatted `String` rather than displaying it in a command window.
- All objects in Java have a `toString` method that returns a `String` representation of the object. Method `toString` is called implicitly when an object appears in code where a `String` is needed.

Section 8.3 Controlling Access to Members

- The access modifiers `public` and `private` control access to a class's variables and methods.

- The primary purpose of `public` methods is to present to the class's clients a view of the services the class provides. Clients need not be concerned with how the class accomplishes its tasks.
- A class's `private` variables and `private` methods (i.e., its implementation details) are not accessible to its clients.

Section 8.4 Referring to the Current Object's Members with the `this` Reference

- A non-`static` method of an object implicitly uses keyword `this` (p. 317) to refer to the object's instance variables and other methods. Keyword `this` can also be used explicitly.
- The compiler produces a separate file with the `.class` extension for every compiled class.
- If a local variable has the same name as a class's field, the local variable shadows the field. You can use the `this` reference in a method to refer to the shadowed field explicitly.

Section 8.5 `Time` Class Case Study: Overloaded Constructors

- Overloaded constructors enable objects of a class to be initialized in different ways. The compiler differentiates overloaded constructors (p. 320) by their signatures.
- To call one contructor of a class from another of the same class, you can use the `this` keyword followed by parentheses containing the constructor arguments. Such a constructor call must appear as the first statement in the constructor's body.

Section 8.6 Default and No-Argument Constructors

- If no constructors are provided in a class, the compiler creates a default constructor.
- If a class declares constructors, the compiler will not create a default constructor. In this case, you must declare a no-argument constructor (p. 322) if default initialization is required.

Section 8.7 Notes on Set *and* Get *Methods*

- *Set* methods are commonly called mutator methods (p. 327) because they typically change a value. *Get* methods are commonly called accessor methods (p. 327) or query methods. A predicate method (p. 327) tests whether a condition is true or false.

Section 8.8 Composition

- A class can have references to objects of other classes as members. This is called composition (p. 328) and is sometimes referred to as a *has-a* relationship.

Section 8.9 Enumerations

- All `enum` types (p. 331) are reference types. An `enum` type is declared with an `enum` declaration, which is a comma-separated list of `enum` constants. The declaration may optionally include other components of traditional classes, such as constructors, fields and methods.
- `enum` constants are implicitly `final`, because they declare constants that should not be modified.
- `enum` constants are implicitly `static`.
- Any attempt to create an object of an `enum` type with operator `new` results in a compilation error.
- `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and to control enhanced `for` statements.
- Each `enum` constant in an `enum` declaration is optionally followed by arguments which are passed to the `enum` constructor.
- For every `enum`, the compiler generates a `static` method called `values` (p. 332) that returns an array of the `enum`'s constants in the order in which they were declared.
- `EnumSet` `static` method `range` (p. 333) receives the first and last `enum` constants in a range and returns an `EnumSet` that contains all the constants between these two constants, inclusive.

Section 8.10 Garbage Collection and Method `finalize`

- Every class in Java has the methods of class `Object`, one of which is the `finalize` method.
- The Java Virtual Machine (JVM) performs automatic garbage collection (p. 334) to reclaim the memory occupied by objects that are no longer in use. When there are no more references to an object, the object is eligible for garbage collection. The memory for such an object can be reclaimed when the JVM executes its garbage collector.
- The `finalize` method (p. 334) is called by the garbage collector just before it reclaims the object's memory. Method `finalize` does not take parameters and has return type `void`.
- The garbage collector (p. 334) may never execute before a program terminates. Thus, it's unclear whether, or when, method `finalize` will be called.

Section 8.11 `static` Class Members

- A `static` variable (p. 334) represents classwide information that's shared among the class's objects.
- Static variables have class scope. A class's `public` `static` members can be accessed through a reference to any object of the class, or they can be accessed by qualifying the member name with the class name and a dot (`.`). Client code can access a class's `private` `static` class members only through methods of the class.
- `static` class members exist as soon as the class is loaded into memory.
- A method declared `static` cannot access non-`static` class members, because a `static` method can be called even when no objects of the class have been instantiated.
- The `this` reference cannot be used in a `static` method.

Section 8.12 `static` Import

- A `static` import declaration (p. 338) enables you to refer to imported `static` members without the class name and a dot (`.`). A single `static` import declaration imports one `static` member, and a `static` import on demand imports all `static` members of a class.

Section 8.13 `final` Instance Variables

- In the context of an application, the principle of least privilege (p. 339) states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task.
- Keyword `final` specifies that a variable is not modifiable. Such variables must be initialized when they're declared or by each of a class's constructors.

Section 8.14 `Time` Class Case Study: Creating Packages

- Each class in the Java API belongs to a package that contains a group of related classes. Packages help manage the complexity of application components and facilitate software reuse.
- Packages provide a convention for unique class names that helps prevent class-name conflicts (p. 342).
- Before a class can be imported into multiple applications, it must be placed in a package. There can be only one `package` declaration (p. 340) in each Java source-code file, and it must precede all other declarations and statements in the file.
- Every package name should start with your Internet domain name in reverse order. After the domain name is reversed, you can choose any other names you want for your package.
- When compiling a class in a package, the `javac` command-line option `-d` (p. 342) specifies where to store the package and causes the compiler to create the package's directories if they do not exist.
- The `package` name is part of the fully qualified class name (p. 342). This helps prevent name conflicts.

- A single-type-import declaration (p. 344) specifies one class to import. A type-import-on-demand declaration (p. 344) imports only the classes that the program uses from a particular package.
- The compiler uses a class loader (p. 344) to locate the classes it needs in the classpath. The classpath consists of a list of directories or archive files, each separated by a directory separator (p. 344).
- The classpath for the compiler and JVM can be specified by providing the `-classpath` option (p. 345) to the `javac` or `java` command, or by setting the `CLASSPATH` environment variable. If classes must be loaded from the current directory, include a dot (`.`) in the classpath.

Section 8.15 Package Access

- If no access modifier is specified for a method or variable when it's declared in a class, the method or variable is considered to have package access (p. 345).

Self-Review Exercise

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

a) When compiling a class in a package, the `javac` command-line option _____ specifies where to store the package and causes the compiler to create the package's directories if they do not exist.
b) `String` class `static` method _____ is similar to method `System.out.printf`, but returns a formatted `String` rather than displaying a `String` in a command window.
c) If a method contains a local variable with the same name as one of its class's fields, the local variable _____ the field in that method's scope.
d) The _____ method is called by the garbage collector just before it reclaims an object's memory.
e) A(n) _____ declaration specifies one class to import.
f) If a class declares constructors, the compiler will not create a(n) _____.
g) An object's _____ method is called implicitly when an object appears in code where a `String` is needed.
h) *Get* methods are commonly called _____ or _____.
i) A(n) _____ method tests whether a condition is true or false.
j) For every `enum`, the compiler generates a `static` method called _____ that returns an array of the `enum`'s constants in the order in which they were declared.
k) Composition is sometimes referred to as a(n) _____ relationship.
l) A(n) _____ declaration contains a comma-separated list of constants.
m) A(n) _____ variable represents classwide information that's shared by all the objects of the class.
n) A(n) _____ declaration imports one `static` member.
o) The _____ states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task.
p) Keyword _____ specifies that a variable is not modifiable.
q) There can be only one _____ in a Java source-code file, and it must precede all other declarations and statements in the file.
r) A(n) _____ declaration imports only the classes that the program uses from a particular package.
s) The compiler uses a(n) _____ to locate the classes it needs in the classpath.
t) The classpath for the compiler and JVM can be specified with the _____ option to the `javac` or `java` command, or by setting the _____ environment variable.
u) *Set* methods are commonly called _____ because they typically change a value.
v) A(n) _____ imports all `static` members of a class.
w) The `public` methods of a class are also known as the class's _____ or _____.

Answers to Self-Review Exercise

8.1 a) `-d`. b) `format`. c) shadows. d) `finalize`. e) single-type-import. f) default constructor. g) `toString`. h) accessor methods, query methods. i) predicate. j) `values`. k) *has-a*. l) `enum`. m) `static`. n) single `static` import. o) principle of least privilege. p) `final`. q) `package` declaration. r) type-import-on-demand. s) class loader. t) `-classpath`, `CLASSPATH`. u) mutator methods. v) `static` import on demand. w) `public` services, `public` interface.

Exercises

8.2 Explain the notion of package access in Java. Explain the negative aspects of package access.

8.3 What happens when a return type, even `void`, is specified for a constructor?

8.4 ***(Rectangle Class)*** Create a class `Rectangle` with attributes `length` and `width`, each of which defaults to 1. Provide methods that calculate the rectangle's perimeter and area. 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`.

8.5 ***(Modifying the Internal Data Representation of a Class)*** It would be perfectly reasonable for the `Time2` class of Fig. 8.5 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 `Time2` class of Fig. 8.5 to implement the `Time2` as the number of seconds since midnight and show that no change is visible to the clients of the class.

8.6 ***(Savings Account Class)*** Create class `SavingsAccount`. Use a `static` variable `annualInterestRate` to store the annual interest rate 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 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 for each of 12 months and print the new balances for both savers. Next, set the `annualInterestRate` to 5%, calculate the next month's interest and print the new balances for both savers.

8.7 ***(Enhancing Class `Time2`)*** Modify class `Time2` of Fig. 8.5 to include a `tick` method that increments the time stored in a `Time2` object by one second. Provide method `incrementMinute` to increment the minute by one and method `incrementHour` to increment the hour by one. Write a 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 and
c) incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

8.8 ***(Enhancing Class `Date`)*** Modify class `Date` of Fig. 8.7 to perform error checking on the initializer values for instance variables `month`, `day` and `year` (currently it validates only the month and day). Provide a method `nextDay` to increment the day by one. Write a program that tests method `nextDay` in a loop that prints the date during each iteration to illustrate that the method works correctly. Test the following cases:

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

8.9 Rewrite the code in Fig. 8.14 to use a separate `import` declaration for each `static` member of class `Math` that's used in the example.

8.10 Write an `enum` type `TrafficLight`, whose constants (`RED`, `GREEN`, `YELLOW`) take one parameter—the duration of the light. Write a program to test the `TrafficLight` `enum` so that it displays the `enum` constants and their durations.

8.11 ***(Complex Numbers)*** Create a class called `Complex` for performing arithmetic with complex numbers. Complex numbers have the form

realPart + *imaginaryPart* * *i*

where *i* is

$\sqrt{-1}$

Write a program to test your class. Use floating-point variables to represent the `private` data of the class. Provide a constructor that enables an object of this class to be initialized when it's declared. Provide a no-argument constructor with default values in case no initializers are provided. Provide `public` methods that perform the following operations:

a) Add two `Complex` numbers: The real parts are added together and the imaginary parts are added together.
b) Subtract 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) Print `Complex` numbers in the form (*realPart*, *imageinaryPart*).

8.12 ***(Date and Time Class)*** Create class `DateAndTime` that combines the modified `Time2` class of Exercise 8.7 and the modified `Date` class of Exercise 8.8. Modify method `incrementHour` to call method `nextDay` 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 program to test the new class `DateAndTime`. Specifically, test incrementing the time to the next day.

8.13 ***(Set of Integers)*** Create class `IntegerSet`. Each `IntegerSet` object can hold integers in the range 0–100. The set is represented by 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 the array to the "empty set" (i.e., all `false` values).

Provide the following methods: The `static` method `union` creates a set that's the set-theoretic union of two existing sets (i.e., an element of the new set's array is set to `true` if that element is `true` in either or both of the existing sets—otherwise, the new set's element is set to `false`). The `static` method `intersection` creates a set which is the set-theoretic intersection of two existing sets (i.e., an element of the new set's array is set to `false` if that element is `false` in either or both of the existing sets—otherwise, the new set's element 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 `toString` returns a `String` containing a set as a list of numbers separated by spaces. Include only those elements that are present in the set. Use `---` to represent an empty set. Method `isEqualTo` determines whether two sets are equal. Write a program to test class `IntegerSet`. Instantiate several `IntegerSet` objects. Test that all your methods work properly.

8.14 ***(Date Class)*** Create class `Date` 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). In the first case the constructor should receive three integer values. In the second case it should receive a `String` and two integer values. In the third case it should receive two integer values, the first of which represents the day number in the year.

[*Hint:* To convert the `String` representation of the month to a numeric value, compare `String`s using the `equals` method. For example, if `s1` and `s2` are `String`s, the method call `s1.equals( s2 )` returns `true` if the `String`s are identical and otherwise returns `false`.]

8.15 ***(Rational Numbers)*** Create a class called `Rational` for performing arithmetic with fractions. Write a 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 that enables an object of this class to be initialized when it's declared. The constructor should store the fraction in reduced form. The fraction

```
2/4
```

is equivalent to `1/2` and 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 that perform each of the following operations:

a) Add two `Rational` numbers: The result of the addition should be stored in reduced form. Implement this as a `static` method.
b) Subtract two `Rational` numbers: The result of the subtraction should be stored in reduced form. Implement this as a `static` method.
c) Multiply two `Rational` numbers: The result of the multiplication should be stored in reduced form. Implement this as a `static` method.
d) Divide two `Rational` numbers: The result of the division should be stored in reduced form. Implement this as a `static` method.
e) Return a `String` representation of a `Rational` number in the form `a/b`, where `a` is the `numerator` and `b` is the `denominator`.
f) Return a `String` representation of a `Rational` number 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.)

8.16 ***(Huge Integer Class)*** Create a class `HugeInteger` which uses a 40-element array of digits to store integers as large as 40 digits each. Provide methods `parse`, `toString`, `add` and `subtract`. Method `parse` should receive a `String`, extract each digit using method `charAt` and place the integer equivalent of each digit into the integer array. For comparing `HugeInteger` objects, provide the following methods: `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` and `isLessThanOrEqualTo`. Each of these is a predicate method that returns `true` if the relationship holds between the two `HugeInteger` objects and returns `false` if the relationship does not hold. Provide a predicate method `isZero`. If you feel ambitious, also provide methods `multiply`, `divide` and `remainder`. [*Note:* Primitive `boolean` values can be output as the word "true" or the word "false" with format specifier `%b`.]

8.17 ***(Tic-Tac-Toe)*** Create a class `TicTacToe` that will enable you to write a program to play Tic-Tac-Toe. The class contains a private 3-by-3 two-dimensional array. Use an enumeration to represent the value in each cell of the array. The enumeration's constants should be named `X`, `O` and `EMPTY` (for a position that does not contain an `X` or an `O`). The constructor should initialize the board elements to `EMPTY`. Allow two human players. Wherever the first player moves, place an `X` in the specified square, and place an `O` 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 it's a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. 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 an extremely challenging project!].

Making a Difference

8.18 ***(Project: Emergency Response Class)*** The North American emergency response service, *9-1-1*, connects callers to a *local* Public Service Answering Point (PSAP). Traditionally, the PSAP would ask the caller for identification information—including the caller's address, phone number and the nature of the emergency, then dispatch the appropriate emergency responders (such as the police, an ambulance or the fire department). *Enhanced 9-1-1 (or E9-1-1)* uses computers and databases to determine the caller's physical address, directs the call to the nearest PSAP, and displays the caller's phone number and address to the call taker. *Wireless Enhanced 9-1-1* provides call takers with identification information for wireless calls. Rolled out in two phases, the first phase required carriers to provide the wireless phone number and the location of the cell site or base station transmitting the call. The second phase required carriers to provide the location of the caller (using technologies such as GPS). To learn more about 9-1-1, visit `www.fcc.gov/pshs/services/911-services/Welcome.html` and `people.howstuffworks.com/9-1-1.htm`.

An important part of creating a class is determining the class's attributes (instance variables). For this class design exercise, research 9-1-1 services on the Internet. Then, design a class called `Emergency` that might be used in an object-oriented 9-1-1 emergency response system. List the attributes that an object of this class might use to represent the emergency. For example, the class might include information on who reported the emergency (including their phone number), the location of the emergency, the time of the report, the nature of the emergency, the type of response and the status of the response. The class attributes should completely describe the nature of the problem and what's happening to resolve that problem.