Software Engineering Observation 8.11

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

Testing and Debugging Tip 8.1

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

8.5 Creating Packages

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

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

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

Common Programming Error 8.4

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

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

To satisfy *Step 2*, we added a **`package`** statement at the beginning of the file. Line 3 uses a **`package`** *statement* to define a **`package`** named **`com.deitel.jhtp4.ch08`**. Placing a **`package`** statement at the beginning of a Java source file indicates that the class defined in the file is part of the specified package. The only types of statements in Java that can appear outside the braces of a class definition are **`package`** statements and **`import`** statements.

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

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

Software Engineering Observation 8.12

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

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

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

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

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

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

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

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

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

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

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

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

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

Common Programming Error 8.5

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

Software Engineering Observation 8.13

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

Testing and Debugging Tip 8.2

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

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

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

8.6 Initializing Class Objects: Constructors

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