Chapter 8 Classes and Objects: A Deeper Look

Java™ How to Program, 8/e

8.1 Introduction

- Deeper look at building classes, controlling access to members of a class and creating constructors.
- Composition—a capability that allows a class to have references to objects of other classes as members.
- More details on `enum` types.
- Discuss `final` instance variables
- Show how to organize classes in packages to help manage large applications and promote reuse.

8.2 `Time` Class Case Study

- Class `Time1` represents the time of day.
- `private int` instance variables `hour`, `minute` and `second` represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23).
- `public` methods `setTime`, `toUniversalString` and `toString`.
  - Called the `public` services or the `public` interface that the class provides to its clients.

```
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; ensure that
11    // the data remains consistent by setting invalid values to zero
12    public void setTime( int h, int m, int s )
13    {
14       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
15       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
16       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
17    } // end method setTime
18
19    // convert to String in universal-time format (HH:MM:SS)
20    public String toUniversalString()
21    {
22       return String.format( "%02d:%02d:%02d", hour, minute, second );
23    } // end method toUniversalString
24
```

Instance variables represent the time in 24-hour clock format

Validate the initial time values

Format the time in 24-hour clock format

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format. (Part 1 of 2.)

```
25      // convert to String in standard-time format (H:MM:SS AM or PM)
26      public String toString()
27      {
28         return String.format( "%d:%02d:%02d %s",
29            ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30            minute, second, ( hour < 12 ? "AM" : "PM" ) );
31      } // end method toString
32   } // end class Time1
```

Format the time in 12-hour clock format; this is also the default `String` format for `Time1`

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format. (Part 2 of 2.)

```
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
```

Create default `Time1` object

Get 24-hour format `String` representation of time

Get 12-hour format `String`; call to `toString` is unnecessary

Set the time using valid values for the hour, minute and second

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

```
25
26          // set time with invalid values; output updated time
27          time.setTime( 99, 99, 99 );
28          System.out.println( "After attempting invalid settings:" );
29          System.out.print( "Universal time: " );
30          System.out.println( time.toUniversalString() );
31          System.out.print( "Standard time: " );
32          System.out.println( time.toString() );
33       } // end main
34    } // end class Time1Test
```

Set the time using invalid values for the hour, minute and second

```
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

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

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

8.2 `Time` Class Case Study (Cont.)

- Class `Time1` does not declare a constructor, so the class has a default constructor that is supplied by the compiler.
- Each instance variable implicitly receives the default value `0` for an `int`.
- Instance variables also can be initialized when they are declared in the class body, using the same initialization syntax as with a local variable.

8.2 `Time` Class Case Study (Cont.)

- 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.
- Reasonable for `Time1` to represent the time internally as the number of sec-onds 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.

8.3 Controlling Access to Members

- Access modifiers `public` and `private` control access to a class's variables and methods.
  - Chapter 9 introduces access modifier `protected`.
- `public` methods 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.
- `private` class members are not accessible outside the class.

Common Programming Error 8.1

An attempt by a method that is 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
```

Each of these statements attempts to access data that is `private` to class `Time1`

Fig. 8.3 | Private members of class `Time1` are not accessible. (Part 1 of 2.)

```
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. (Part 2 of 2.)

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

- Every object can access a reference to itself with keyword `this`.
- 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.
  - Enables the class's code to know which object should be manipulated.
  - Can also use keyword `this` explicitly in a non-`static` method's body.
- Can use the `this` reference implicitly and explicitly.

8.4 Referring to the Current Object's Members with the `this` Reference (Cont.)

- 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.
- When one source-code (`.java`) file contains multiple class declarations, the compiler places both class files for those classes in the same directory.
- 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.

```
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
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 3.)

```
20      // if the constructor uses parameter names identical to
21      // instance variable names the "this" reference is
22      // required to distinguish between 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
```

The `this` reference enables you to explicitly access instance variables when they are shadowed by local variables of the same name

The `this` reference is not required to call other methods of the same class

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 2 of 3.)

```
38      // convert to String in universal-time format (HH:MM:SS)
39      public String toUniversalString()
40      {
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" not required here, since the instance variables are not shadowed

```
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 3 of 3.)

8.4 Referring to the Current Object's Members with the `this` Reference (Cont.)

- `SimpleTime` declares three `private` instance variables—`hour`, `minute` and `second`.
- If parameter names for the constructor that are identical to the class's instance-variable names.
  - We don't recommend this practice
  - Use it here to shadow (hide) the corresponding instance
  - Illustrates 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 uses the local variable rather than the field.
  - The local variable *shadows* the field in the method's scope.
- A method can use the `this` reference to refer to the shadowed field explicitly.

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.*

8.5 `Time` Class Case Study: Overloaded Constructors

- Overloaded constructors enable objects of a class to be initialized in different ways.
- To overload constructors, simply provide multiple constructor declarations with different signatures.
- Recall that the compiler differentiates signatures by the *number* of parameters, the *types* of the parameters and the *order* of the parameter types in each signature.

8.5 `Time` Class Case Study: Overloaded Constructors (Cont.)

- Class `Time2` (Fig. 8.5) contains five overloaded constructors that provide convenient ways to initialize objects of the new class `Time2`.
- 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.

8.5 `Time` Class Case Study: Overloaded Constructors (Cont.)

- A program can declare a so-called no-argument constructor that is invoked without arguments.
- Such a constructor simply initializes the object as specified in the constructor's body.
- Using `this` in method-call syntax as the first statement in a constructor's body invokes another constructor of the same class.
  - 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.
- Once you declare any constructors in a class, the compiler will not provide a default constructor.

Common Programming Error 8.4

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

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`*).*

```
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: initializes each instance variable
11    // to zero; ensures that Time2 objects start in a consistent state
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
```

Invoke three-argument constructor

Invoke three-argument constructor

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 1 of 5.)

```
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
```

Invoke three-argument constructor

Invoke `setTime` to validate the data

Invoke three-argument constructor

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 2 of 5.)

```
42      // Set Methods
43      // set a new time value using universal time; ensure that
44      // the data remains consistent by setting invalid values to zero
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         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56      } // end method setHour
57
58      // validate and set minute
59      public void setMinute( int m )
60      {
61         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62      } // end method setMinute
63
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 3 of 5.)

```
64      // validate and set second
65      public void setSecond( int s )
66      {
67         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68      } // end method setSecond
69
70      // Get Methods
71      // get hour value
72      public int getHour()
73      {
74         return hour;
75      } // end method getHour
76
77      // get minute value
78      public int getMinute()
79      {
80         return minute;
81      } // end method getMinute
82
83      // get second value
84      public int getSecond()
85      {
86         return second;
87      } // end method getSecond
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 4 of 5.)

```
88
89      // convert to String in universal-time format (HH:MM:SS)
90      public String toUniversalString()
91      {
92         return String.format(
93            "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
94      } // end method toUniversalString
95
96      // convert to String in standard-time format (H:MM:SS AM or PM)
97      public String toString()
98      {
99         return String.format( "%d:%02d:%02d %s",
100           ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
101           getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
102     } // end method toString
103  } // end class Time2
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 5 of 5.)

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test
5  {
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( 27, 74, 99 ); // 00:00:00
13       Time2 t6 = new Time2( t4 ); // 12:25:42
14
15       System.out.println( "Constructed with:" );
16       System.out.println( "t1: all arguments defaulted" );
17       System.out.printf( "   %s\n", t1.toUniversalString() );
18       System.out.printf( "   %s\n", t1.toString() );
19
20       System.out.println(
21          "t2: hour specified; minute and second defaulted" );
22       System.out.printf( "   %s\n", t2.toUniversalString() );
23       System.out.printf( "   %s\n", t2.toString() );
24
```

Compiler determines which constructor to call based on the number and types of the arguments

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

```
25          System.out.println(
26             "t3: hour and minute specified; second defaulted" );
27          System.out.printf( "   %s\n", t3.toUniversalString() );
28          System.out.printf( "   %s\n", t3.toString() );
29
30          System.out.println( "t4: hour, minute and second specified" );
31          System.out.printf( "   %s\n", t4.toUniversalString() );
32          System.out.printf( "   %s\n", t4.toString() );
33
34          System.out.println( "t5: all invalid values specified" );
35          System.out.printf( "   %s\n", t5.toUniversalString() );
36          System.out.printf( "   %s\n", t5.toString() );
37
38          System.out.println( "t6: Time2 object t4 specified" );
39          System.out.printf( "   %s\n", t6.toUniversalString() );
40          System.out.printf( "   %s\n", t6.toString() );
41       } // end main
42    } // end class Time2Test
```

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

```
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
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: all invalid values specified
   00:00:00
   12:00:00 AM
t6: Time2 object t4 specified
   12:25:42
   12:25:42 PM
```

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

Common Programming Error 8.3

It's a syntax 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 syntax error when a method attempts to invoke a constructor directly via* `this`.

8.5 `Time` Class Case Study: Overloaded Constructors (Cont.)

- Notes regarding class `Time2`'s *set* and *get* methods and constructors
- Methods can access a 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.

8.5 `Time` Class Case Study: Overloaded Constructors (Cont.)

- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.
- Similarly, each `Time2` constructor could be written to include a copy of the appropriate statements from methods `setHour`, `setMinute` and `setSecond`.
  - Doing so may be slightly more efficient, because the extra constructor call and call to `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 any changes to the implementation of `setTime` to be made only once.

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.*

8.6 Default and No-Argument Constructors

- Every class must have at least one constructor.
- If you do not provide any constructors 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).
- 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.

8.7 Notes on *Set and Get Methods*

- Classes often provide `public` methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) `private` instance variables.
- *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.

8.7 Notes on *Set and Get Methods (Cont.)*

- It would seem that providing *set* and *get* capabilities is essentially the same as making the instance variables `public`.
  - A `public` instance variable can be read or written by any method that has a reference to an object that contains 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.
  - A `public` *set* method can control the modifications to the variable's value to ensure that the new value is consistent for that data item.
- Although *set* and *get* methods provide access to `private` data, it is restricted by the implementation of the methods.

8.7 Notes on *Set and Get Methods (Cont.)*

- ***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.
- ***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.
  - Example: `ArrayList`'s `isEmpty` method, which returns `true` if the `ArrayList` is empty.

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.

```
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    // constructor: call checkMonth to confirm proper value for month;
11    // call checkDay to confirm proper value for day
12    public Date( int theMonth, int theDay, int theYear )
13    {
14       month = checkMonth( theMonth ); // validate month
15       year = theYear; // could validate year
16       day = checkDay( theDay ); // validate day
17
18       System.out.printf(
19          "Date object constructor for date %s\n", this );
20    } // end Date constructor
21
```

Fig. 8.7 | `Date` class declaration. (Part 1 of 3.)

```
22      // utility method to confirm proper month value
23      private int checkMonth( int testMonth )
24      {
25         if ( testMonth > 0 && testMonth <= 12 ) // validate month
26            return testMonth;
27         else // month is invalid
28         {
29            System.out.printf(
30               "Invalid month (%d) set to 1.", testMonth );
31            return 1; // maintain object in consistent state
32         } // end else
33      } // end method checkMonth
34
35      // utility method to confirm proper day value based on month and year
36      private int checkDay( int testDay )
37      {
38         int[] daysPerMonth =
39            { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
40
41         // check if day in range for month
42         if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
43            return testDay;
44
```

Fig. 8.7 | `Date` class declaration. (Part 2 of 3.)

```
45         // check for leap year
46         if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
47              ( year % 4 == 0 && year % 100 != 0 ) ) )
48            return testDay;
49
50         System.out.printf( "Invalid day (%d) set to 1.", testDay );
51         return 1;  // maintain object in consistent state
52      } // end method checkDay
53
54      // return a String of the form month/day/year
55      public String toString()
56      {
57         return String.format( "%d/%d/%d", month, day, year );
58      } // end method toString
59   } // end class Date
```

Fig. 8.7 | `Date` class declaration. (Part 3 of 3.)

```
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;
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
```

References to other objects composed into class `Employee`

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

```
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.)

```
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` objects used to initialize `Employee`

Gets `Employee`'s `String` representation by calling `toString` implicitly

```
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

- The basic `enum` type defines a set of constants represented as unique identifiers.
- 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.

8.9 Enumerations (Cont.)

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - `enum` constants are implicitly `final`, because they declare constants that shouldn't 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.
  - `enum` declarations contain two parts—the `enum` constants and the other members of the `enum` type.
  - An `enum` constructor can specify any number of parameters and can be overloaded.
- For every `enum`, the compiler generates the `static` method `values` that returns an array of the `enum`'s constants.
- When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation.

```
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", "2010" ),
9     CHTP( "C How to Program", "2007" ),
10    IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTP( "C++ How to Program", "2008" ),
12    VBHTP( "Visual Basic 2008 How to Program", "2009" ),
13    CSHARPHTP( "Visual C# 2008 How to Program", "2009" );
14
15    // instance fields
16    private final String title; // book title
17    private final String copyrightYear; // copyright year
18
```

enum constants initialized with constructor calls

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

```
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.)

```
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" );
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
```

enum method values returns a collection of the enum constants

EnumSet method range returns a collection of the enum constants in the specified range of constants

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

```
All books:

JHTP      Java How to Program                           2010
CHTP      C How to Program                              2007
IW3HTP    Internet & World Wide Web How to Program      2008
CPPHTP    C++ How to Program                            2008
VBHTP     Visual Basic 2008 How to Program              2009
CSHARPHTP Visual C# 2008 How to Program                 2009

Display a range of enum constants:

JHTP      Java How to Program                           2010
CHTP      C How to Program                              2007
IW3HTP    Internet & World Wide Web How to Program      2008
CPPHTP    C++ How to Program                            2008
```

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

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.
  - Rarely used because it can cause performance problems and there is some uncertainty as to whether it will get called.
- Every object uses system resources, such as memory.
  - Need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur.
- 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.

8.10 Garbage Collection and Method `finalize` (Cont.)

- 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.

8.10 Garbage Collection and Method `finalize` (Cont.)

- 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.
  - The garbage collector may never execute before a program terminates.
  - Thus, it's unclear if, or when, method `finalize` will be called.
  - For this reason, most programmers should avoid method `finalize`.

8.11 `String` objects in Java are immutable

- `String` objects in Java are immutable—they cannot be modified after they are created.
- String-concatenation operations actually result in a new `String` object containing the concatenated values—the original `String` objects are not modified.
- When you use String **s = "abc",** you create a String reference to a String object that has the immutable value "abc".
- Then, when you say **s = s.substring(1);,** you assign s to a newly created String object that contains "bc" - but the original object is unchanged.
- String s = "Test";
- String j = s;
- s = s.substring(1);
- s is now T and j is still Test.

8.13 `final` Instance Variables

- Keyword `final` specifies that a variable is not modifiable (i.e., it's a constant) and any attempt to modify it is an error.

```
private final int INCREMENT;
```

  - Declares a `final` (constant) instance variable `INCREMENT` of type `int`.
- `final` variables can be initialized when they are declared or by each of the class's constructors so that each object of the class has a different value.
- If a class provides multiple constructors, every one would be required to initialize each `final` variable.
- A `final` variable cannot be modified by assignment after it's initialized.

Software Engineering Observation 8.10

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.*

```
1  // Fig. 8.15: Increment.java
2  // final instance variable in a class.
3
4  public class Increment
5  {
6     private int total = 0; // total of all increments
7     private final int INCREMENT; // constant variable (uninitialized)
8
9     // constructor initializes final instance variable INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // initialize constant variable (once)
13    } // end Increment constructor
14
15    // add INCREMENT to total
16    public void addIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // end method addIncrementToTotal
20
```

`final` variable must be initialized

Constructor performs the initialization

Fig. 8.15 | `final` instance variable in a class. (Part 1 of 2.)

```
21      // return String representation of an Increment object's data
22      public String toString()
23      {
24         return String.format( "total = %d", total );
25      } // end method toString
26   } // end class Increment
```

Fig. 8.15 | `final` instance variable in a class. (Part 2 of 2.)

```
1  // Fig. 8.16: IncrementTest.java
2  // final variable initialized with a constructor argument.
3
4  public class IncrementTest
5  {
6     public static void main( String[] args )
7     {
8        Increment value = new Increment( 5 );
9
10       System.out.printf( "Before incrementing: %s\n\n", value );
11
12       for ( int i = 1; i <= 3; i++ )
13       {
14          value.addIncrementToTotal();
15          System.out.printf( "After increment %d: %s\n", i, value );
16       } // end for
17    } // end main
18 } // end class IncrementTest
```

Argument passed to constructor to initialize the `final` instance variable

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Fig. 8.16 | `final` variable initialized with a constructor argument.

Software Engineering Observation 8.11

A `final` *field should also be declared* `static` *if it's initialized in its declaration to a value that is 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.*

Common Programming Error 8.11

Not initializing a `final` *instance variable in its declaration or in every constructor of the class yields a compilation error indicating that the variable might not have been initialized.*

8.15 `Time` Class Case Study: Creating Packages

- Each class in the Java API belongs to a package that contains a group of related classes.
- Packages are defined once, but can be imported into many programs.
- Packages help programmers manage the complexity of application components.
- Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- Packages provide a convention for unique class names, which helps prevent class-name conflicts.

8.15 `Time` Class Case Study: Creating Packages (Cont.)

- The steps for creating a reusable class:
- Declare a `public` class; otherwise, it can be used only by other classes in the same package.
- 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.
- Compile the class so that it's placed in the appropriate package directory.
- Import the reusable class into a program and use the class.

8.15 `Time` Class Case Study: Creating Packages (Cont.)

- 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:
  - a `package` declaration (if any),
  - `import` declarations (if any), then
  - 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.

```
1  // Fig. 8.18: 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; ensure that
12    // the data remains consistent by setting invalid values to zero
13    public void setTime( int h, int m, int s )
14    {
15       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
16       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
17       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
18    } // end method setTime
19
20    // convert to String in universal-time format (HH:MM:SS)
21    public String toUniversalString()
22    {
23       return String.format( "%02d:%02d:%02d", hour, minute, second );
24    } // end method toUniversalString
```

Helps make `Time1` a unique class name; must be first statement in file

Fig. 8.18 | Packaging class `Time1` for reuse. (Part 1 of 2.)

```
25
26      // convert to String in standard-time format (H:MM:SS AM or PM)
27      public String toString()
28      {
29         return String.format( "%d:%02d:%02d %s",
30            ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
31            minute, second, ( hour < 12 ? "AM" : "PM" ) );
32      } // end method toString
33   } // end class Time1
```

Fig. 8.18 | Packaging class `Time1` for reuse. (Part 2 of 2.)

8.15 `Time` Class Case Study: Creating Packages (Cont.)

- 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

```
package com.deitel.jhtp.ch08;
```

- indicates that class `Time1` should be placed in the directory

```
com
   deitel
      jhtp
         ch08
```

- The directory names in the `package` declaration specify the exact location of the classes in the package.

8.15 `Time` Class Case Study: Creating Packages (Cont.)

- The `package` name is part of the fully qualified class name.
  - Class `Time1`'s name is actually `com.deitel.jhtp.ch08.Time1`
- Can use the fully qualified name in programs, or `import` the class and use its simple name (the class name by itself).
- If another package contains a class by the same name, 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).

8.15 `Time` Class Case Study: Creating Packages (Cont.)

- Fig. 8.19, line 3 is a single-type-import declaration
  - It specifies one class to import.
- When your program uses multiple classes from the same package, you can import those classes with a type-import-on-demand declaration.
- Example:

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

- uses an asterisk (*) at the end of the `import` declaration to inform the compiler that all `public` classes from the `java.util` package are available for use in the program.
  - Only the classes from package `java-.util` that are used in the program are loaded by the JVM.

Common Programming Error 8.12

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.*

8.16 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.
- In a program uses multiple classes from the same package, 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.

```
1  // Fig. 8.20: 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
```

Accessing package access variables in class `PackageData`

Fig. 8.20 | Package-access members of a class are accessible by other classes in the same package. (Part 1 of 3.)

```
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
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
```

Class has package access; can be used only by other classes in the same directory

Package access data can be accessed by other classes in the same package via a reference to an object of the class

Fig. 8.20 | Package-access members of a class are accessible by other classes in the same package. (Part 2 of 3.)

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

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

Fig. 8.20 | Package-access members of a class are accessible by other classes in the same package. (Part 3 of 3.)