Lesson 6
Classes and Objects: A Deeper Look

Assoc. Prof. Marenglen Biba

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

- 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 `static` class members and `final` instance variables in detail.
- 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.)

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

- 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.

```
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`.
- Parameter names for the constructor can be 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.

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.

```
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.)

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.

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.
- 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.

```
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.)

8.9 Enumerations (Cont.)

- Use the `static` method `range` of class `EnumSet` (declared in package `java.util`) to access a range of an `enum`'s constants.
  - Method `range` takes two parameters—the first and the last `enum` constants in the range
  - Returns an `EnumSet` that contains all the constants between these two constants, inclusive.
- The enhanced `for` statement can be used with an `EnumSet` just as it can with an array.
- Class `EnumSet` provides several other `static` methods.
  - `java.sun.com/javase/7/docs/api/java/util/EnumSet.html`

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, but some can still happen in subtle ways.
- Other types of resource leaks can occur.
  - 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.

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`.

Software Engineering Observation 8.7

A class that uses system resources, such as files on disk, should provide a method that programmers can call to release resources when they are no longer needed in a program. Many Java API classes provide `close` *or* `dispose` *methods for this purpose. For example, class* `Scanner` *(*`java.sun.com/javase/6/docs/api/java/util/Scanner.html`*) has a* `close` *method.*

8.11 `static` Class Members

- 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`.

8.11 `static` Class Members (Cont.)

- Static variables have class scope.
- 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()`.
- `private` `static` class members can be accessed by client code only through methods of the class.
- `static` class members are 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.

8.11 `static` Class Members (Cont.)

- 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.
- If a `static` variable is not initialized, the compiler assigns it a default value—in this case `0`, the default value for type `int`.

```
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
```

`static` variable shared by all `Employees`

`static` variables can be access by *all* of the class's methods

Fig. 8.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 1 of 2.)

```
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
```

`static` method can be called by the class's clients to get the current count—whether or not there are any `Employee` objects in memory

Fig. 8.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 2 of 2.)

```
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
```

Gets the count before creating `Employees`

Gets the count after creating `Employees`; should call `static` methods only via the class name

Gets the count after creating `Employees`

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

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.

```
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
```

Good practice to set variables to `null` when you no longer need the objects they refer to; enables the garbage collector to retrieve them if there are no other references to those objects.

```
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.)

8.11 `static` Class Members (Cont.)

- Objects become "eligible for garbage collection" when there are no more references to them in the program.
- 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 the garbage collector does execute, it's possible that no objects or only a subset of the eligible objects will be collected.

8.12 `static` Import

- 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.
- Two forms
  - One that imports a particular `static` member (which is known as single `static` import)
  - One that imports all `static` members of a class (which is known as `static` import on demand)

8.12 `static` Import (Cont.)

- The following syntax imports a particular `static` member:

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

- where *packageName* is the package of the class, ClassName is the name of the class and *staticMemberName* is the name of the `static` field or method.
- The following syntax imports all `static` members of a class:

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

- where *packageName* is the package of the class and *ClassName* is the name of the class.
  - * indicates that *all* `static` members of the specified class should be available for use in the class(es) declared 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.

```
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( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // end main
14 } // end class StaticImportTest
```

Enables `Math` methods to be used by their simple names in this file

```
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.
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.
  - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.
- 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`.

8.14 `final` Instance Variables

- `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.

```
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.

8.14 `final` Instance Variables (Cont.)

- If a `final` variable is not initialized, a compilation error occurs.

```
Increment.java:13: variable INCREMENT might not have been initialized
   } // end Increment constructor
   ^
1 error
```

Fig. 8.17 | `final` variable `INCREMENT` must be 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.

```
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.)

- `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.
- Example:

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

- specifies that the first directory in our package name should be placed in the current directory (`.`).
- The compiled classes are placed into the directory that is named last in the `package` statement.

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).

```
1  // Fig. 8.19: 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(); // calls 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() );
```

Imports class `Time1` for use in this source code file

Fig. 8.19 | `Time1` object used in an application. (Part 1 of 2.)

```
25         System.out.println(); // output a blank line
26
27         // set time with invalid values; output updated time
28         time.setTime( 99, 99, 99 );
29         System.out.println( "After attempting invalid settings:" );
30         System.out.print( "Universal time: " );
31         System.out.println( time.toUniversalString() );
32         System.out.print( "Standard time: " );
33         System.out.println( time.toString() );
34      } // end main
35   } // end class Time1PackageTest
```

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

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.

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

- Specifying the Classpath During Compilation
- When compiling a class that uses classes from other packages, `javac` must locate the `.class` files for all other classes being used.
- 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.
  - 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.

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

- The classpath consists of a list of directories or archive files, each separated by a directory separator
  - 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.
  - 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.

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

- By default, the classpath consists only of the current directory.
- The classpath can be modified by:
  - providing the `-classpath` option to the `javac` compiler
  - setting the `CLASSPATH` environment variable (not recommended).
- Classpath
  - http://docs.oracle.com/javase/7/docs/technotes/tools/index.html#genera
  - The section entitled "General Information" contains information on setting the classpath for UNIX/Linux and Windows.

Common Programming Error 8.13

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 8.12

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.*

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

- 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.
  - If classes must be loaded from the current directory, be sure to include a dot (`.`) in the classpath to specify the current directory.

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.
- If 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.)

End of Part I

- Chapter 8
  - Java™ How to Program, 9/e