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.

```java
// Fig. 8.1: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void setTime( int h, int m, int s )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
      second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
   } // end method setTime

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      return String.format( "%02d:%02d:%02d", hour, minute, second );
   } // end method toUniversalString

```

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.)

```java
   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString()
   {
      return String.format( "%d:%02d:%02d %s",
         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
         minute, second, ( hour < 12 ? "AM" : "PM" ) );
   } // end method toString
} // 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.)

```java
// Fig. 8.2: Time1Test.java
// Time1 object used in an application.

public class Time1Test
{
   public static void main( String[] args )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      System.out.print( "The initial universal time is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "The initial standard time is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // change time and output updated time
      time.setTime( 13, 27, 6 );
      System.out.print( "Universal time after setTime is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time after setTime is: " );
      System.out.println( time.toString() );
      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.)

```java

      // set time with invalid values; output updated time
      time.setTime( 99, 99, 99 );
      System.out.println( "After attempting invalid settings:" );
      System.out.print( "Universal time: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time: " );
      System.out.println( time.toString() );
   } // end main
} // 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.

```java
// Fig. 8.3: MemberAccessTest.java
// Private members of class Time1 are not accessible.
public class MemberAccessTest
{
   public static void main( String[] args )
   {
      Time1 time = new Time1(); // create and initialize Time1 object

      time.hour = 7; // error: hour has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
      time.second = 30; // error: second has private access in Time1
   } // end main
} // 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.

```java
// Fig. 8.4: ThisTest.java
// this used implicitly and explicitly to refer to members of an object.

public class ThisTest
{
   public static void main( String[] args )
   {
      SimpleTime time = new SimpleTime( 15, 30, 19 );
      System.out.println( time.buildString() );
   } // end main
} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime
{
   private int hour; // 0-23
   private int minute; // 0-59
   private int second; // 0-59

```

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

```java
   // if the constructor uses parameter names identical to
   // instance variable names the "this" reference is
   // required to distinguish between names
   public SimpleTime( int hour, int minute, int second )
   {
      this.hour = hour; // set "this" object's hour
      this.minute = minute; // set "this" object's minute
      this.second = second; // set "this" object's second
   } // end SimpleTime constructor

   // use explicit and implicit "this" to call toUniversalString
   public String buildString()
   {
      return String.format( "%24s: %s\n%24s: %s",
         "this.toUniversalString()", this.toUniversalString(),
         "toUniversalString()", toUniversalString() );
   } // end method buildString

```

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.)

```java
   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      // "this" is not required here to access instance variables,
      // because method does not have local variables with same
      // names as instance variables
      return String.format( "%02d:%02d:%02d",
         this.hour, this.minute, this.second );
   } // end method toUniversalString
} // 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.

```java
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded constructors.

public class Time2
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor: initializes each instance variable
   // to zero; ensures that Time2 objects start in a consistent state
   public Time2()
   {
      this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 no-argument constructor

   // Time2 constructor: hour supplied, minute and second defaulted to 0
   public Time2( int h )
   {
      this( h, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 one-argument constructor

```

Invoke three-argument constructor

Invoke three-argument constructor

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

```java
   // Time2 constructor: hour and minute supplied, second defaulted to 0
   public Time2( int h, int m )
   {
      this( h, m, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 two-argument constructor

   // Time2 constructor: hour, minute and second supplied
   public Time2( int h, int m, int s )
   {
      setTime( h, m, s ); // invoke setTime to validate time
   } // end Time2 three-argument constructor

   // Time2 constructor: another Time2 object supplied
   public Time2( Time2 time )
   {
      // invoke Time2 three-argument constructor
      this( time.getHour(), time.getMinute(), time.getSecond() );
   } // end Time2 constructor with a Time2 object argument

```

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.)

```java
   // Set Methods
   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void setTime( int h, int m, int s )
   {
      setHour( h ); // set the hour
      setMinute( m ); // set the minute
      setSecond( s ); // set the second
   } // end method setTime

   // validate and set hour
   public void setHour( int h )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
   } // end method setHour

   // validate and set minute
   public void setMinute( int m )
   {
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
   } // end method setMinute

```

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

```
   // validate and set second
   public void setSecond( int s )
   {
      second = ( ( s >= 0 && s < 60 ) ? s : 0 );
   } // end method setSecond

   // Get Methods
   // get hour value
   public int getHour()
   {
      return hour;
   } // end method getHour

   // get minute value
   public int getMinute()
   {
      return minute;
   } // end method getMinute

   // get second value
   public int getSecond()
   {
      return second;
   } // end method getSecond
```

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

```java

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      return String.format(
         "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
   } // end method toUniversalString

   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString()
   {
      return String.format( "%d:%02d:%02d %s",
         ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
         getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
   } // end method toString
} // end class Time2
```

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

```java
// Fig. 8.6: Time2Test.java
// Overloaded constructors used to initialize Time2 objects.

public class Time2Test
{
   public static void main( String[] args )
   {
      Time2 t1 = new Time2(); // 00:00:00
      Time2 t2 = new Time2( 2 ); // 02:00:00
      Time2 t3 = new Time2( 21, 34 ); // 21:34:00
      Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
      Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
      Time2 t6 = new Time2( t4 ); // 12:25:42

      System.out.println( "Constructed with:" );
      System.out.println( "t1: all arguments defaulted" );
      System.out.printf( "   %s\n", t1.toUniversalString() );
      System.out.printf( "   %s\n", t1.toString() );

      System.out.println(
         "t2: hour specified; minute and second defaulted" );
      System.out.printf( "   %s\n", t2.toUniversalString() );
      System.out.printf( "   %s\n", t2.toString() );

```

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.)

```
      System.out.println(
         "t3: hour and minute specified; second defaulted" );
      System.out.printf( "   %s\n", t3.toUniversalString() );
      System.out.printf( "   %s\n", t3.toString() );

      System.out.println( "t4: hour, minute and second specified" );
      System.out.printf( "   %s\n", t4.toUniversalString() );
      System.out.printf( "   %s\n", t4.toString() );

      System.out.println( "t5: all invalid values specified" );
      System.out.printf( "   %s\n", t5.toUniversalString() );
      System.out.printf( "   %s\n", t5.toString() );

      System.out.println( "t6: Time2 object t4 specified" );
      System.out.printf( "   %s\n", t6.toUniversalString() );
      System.out.printf( "   %s\n", t6.toString() );
   } // end main
} // 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.

```java
// Fig. 8.7: Date.java
// Date class declaration.

public class Date
{
   private int month; // 1-12
   private int day; // 1-31 based on month
   private int year; // any year

   // constructor: call checkMonth to confirm proper value for month;
   // call checkDay to confirm proper value for day
   public Date( int theMonth, int theDay, int theYear )
   {
      month = checkMonth( theMonth ); // validate month
      year = theYear; // could validate year
      day = checkDay( theDay ); // validate day

      System.out.printf(
         "Date object constructor for date %s\n", this );
   } // end Date constructor

```

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

```java
   // utility method to confirm proper month value
   private int checkMonth( int testMonth )
   {
      if ( testMonth > 0 && testMonth <= 12 ) // validate month
         return testMonth;
      else // month is invalid
      {
         System.out.printf(
            "Invalid month (%d) set to 1.", testMonth );
         return 1; // maintain object in consistent state
      } // end else
   } // end method checkMonth

   // utility method to confirm proper day value based on month and year
   private int checkDay( int testDay )
   {
      int[] daysPerMonth =
         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

      // check if day in range for month
      if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
         return testDay;

```

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

```
      // check for leap year
      if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
           ( year % 4 == 0 && year % 100 != 0 ) ) )
         return testDay;

      System.out.printf( "Invalid day (%d) set to 1.", testDay );
      return 1;  // maintain object in consistent state
   } // end method checkDay

   // return a String of the form month/day/year
   public String toString()
   {
      return String.format( "%d/%d/%d", month, day, year );
   } // end method toString
} // end class Date
```

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

```java
// Fig. 8.8: Employee.java
// Employee class with references to other objects.

public class Employee
{
   private String firstName;
   private String lastName;
   private Date birthDate;
   private Date hireDate;

   // constructor to initialize name, birth date and hire date
   public Employee( String first, String last, Date dateOfBirth,
      Date dateOfHire )
   {
      firstName = first;
      lastName = last;
      birthDate = dateOfBirth;
      hireDate = dateOfHire;
   } // end Employee constructor

```

References to other objects composed into class `Employee`

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

```
   // convert Employee to String format
   public String toString()
   {
      return String.format( "%s, %s  Hired: %s  Birthday: %s",
         lastName, firstName, hireDate, birthDate );
   } // end method toString
} // end class Employee
```

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

```java
// Fig. 8.9: EmployeeTest.java
// Composition demonstration.

public class EmployeeTest
{
   public static void main( String[] args )
   {
      Date birth = new Date( 7, 24, 1949 );
      Date hire = new Date( 3, 12, 1988 );
      Employee employee = new Employee( "Bob", "Blue", birth, hire );

      System.out.println( employee );
   } // end main
} // 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.

```java
// Fig. 8.10: Book.java
// Declaring an enum type with constructor and explicit instance fields
// and accessors for these fields

public enum Book
{
   // declare constants of enum type
   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" );

   // instance fields
   private final String title; // book title
   private final String copyrightYear; // copyright year

```

`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.)

```
   // enum constructor
   Book( String bookTitle, String year )
   {
      title = bookTitle;
      copyrightYear = year;
   } // end enum Book constructor

   // accessor for field title
   public String getTitle()
   {
      return title;
   } // end method getTitle

   // accessor for field copyrightYear
   public String getCopyrightYear()
   {
      return copyrightYear;
   } // end method getCopyrightYear
} // 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.)

```java
// Fig. 8.11: EnumTest.java
// Testing enum type Book.
import java.util.EnumSet;

public class EnumTest
{
   public static void main( String[] args )
   {
      System.out.println( "All books:\n" );

      // print all books in enum Book
      for ( Book book : Book.values() )
         System.out.printf( "%-10s%-45s%s\n", book,
            book.getTitle(), book.getCopyrightYear() );

      System.out.println( "\nDisplay a range of enum constants:\n" );

      // print first four books
      for ( Book book : EnumSet.range( Book.JHTP, Book.CPPHTP ) )
         System.out.printf( "%-10s%-45s%s\n", book,
            book.getTitle(), book.getCopyrightYear() );
   } // end main
} // 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`.

```java
// Fig. 8.12: Employee.java
// Static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
   private String firstName;
   private String lastName;
   private static int count = 0; // number of Employees created

   // initialize Employee, add 1 to static count and
   // output String indicating that constructor was called
   public Employee( String first, String last )
   {
      firstName = first;
      lastName = last;

      ++count;  // increment static count of employees
      System.out.printf( "Employee constructor: %s %s; count = %d\n",
         firstName, lastName, count );
   } // end Employee constructor

```

`static` variable shared by all `Employees`

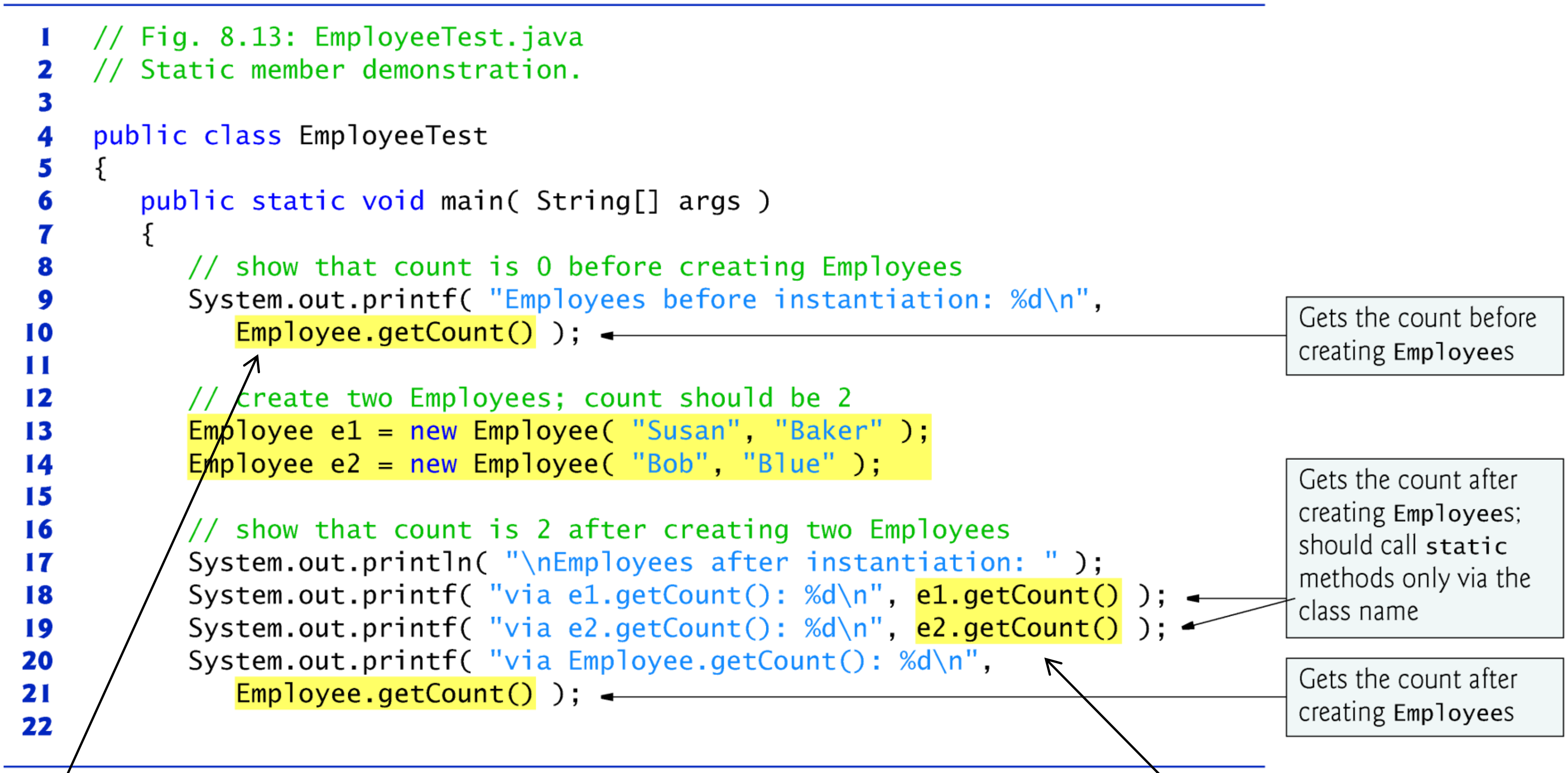
`static` variables can be access by *all* of the class's methods

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

```java
   // get first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // get last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // static method to get static count value
   public static int getCount()
   {
      return count;
   } // end method getCount
} // 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.)



```
// Fig. 8.13: EmployeeTest.java
// Static member demonstration.

public class EmployeeTest
{
   public static void main( String[] args )
   {
      // show that count is 0 before creating Employees
      System.out.printf( "Employees before instantiation: %d\n",
         Employee.getCount() );

      // create two Employees; count should be 2
      Employee e1 = new Employee( "Susan", "Baker" );
      Employee e2 = new Employee( "Bob", "Blue" );

      // show that count is 2 after creating two Employees
      System.out.println( "\nEmployees after instantiation: " );
      System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
      System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
      System.out.printf( "via Employee.getCount(): %d\n",
         Employee.getCount() );

```

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.

```
      // get names of Employees
      System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
         e1.getFirstName(), e1.getLastName(),
         e2.getFirstName(), e2.getLastName() );

      // in this example, there is only one reference to each Employee,
      // so the following two statements indicate that these objects
      // are eligible for garbage collection
      e1 = null;
      e2 = null;
   } // end main
} // 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.

8.15 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.

```java
// Fig. 8.18: Time1.java
// Time1 class declaration maintains the time in 24-hour format.
package com.deitel.jhtp.ch08;

public class Time1
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void setTime( int h, int m, int s )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
      second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
   } // end method setTime

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      return String.format( "%02d:%02d:%02d", hour, minute, second );
   } // 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.)

```java

   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString()
   {
      return String.format( "%d:%02d:%02d %s",
         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
         minute, second, ( hour < 12 ? "AM" : "PM" ) );
   } // end method toString
} // 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).

```java
// Fig. 8.19: Time1PackageTest.java
// Time1 object used in an application.
import com.deitel.jhtp.ch08.Time1; // import class Time1

public class Time1PackageTest
{
   public static void main( String[] args )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // calls Time1 constructor

      // output string representations of the time
      System.out.print( "The initial universal time is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "The initial standard time is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // change time and output updated time
      time.setTime( 13, 27, 6 );
      System.out.print( "Universal time after setTime is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time after setTime is: " );
      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.)

```java
      System.out.println(); // output a blank line

      // set time with invalid values; output updated time
      time.setTime( 99, 99, 99 );
      System.out.println( "After attempting invalid settings:" );
      System.out.print( "Universal time: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time: " );
      System.out.println( time.toString() );
   } // end main
} // 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.

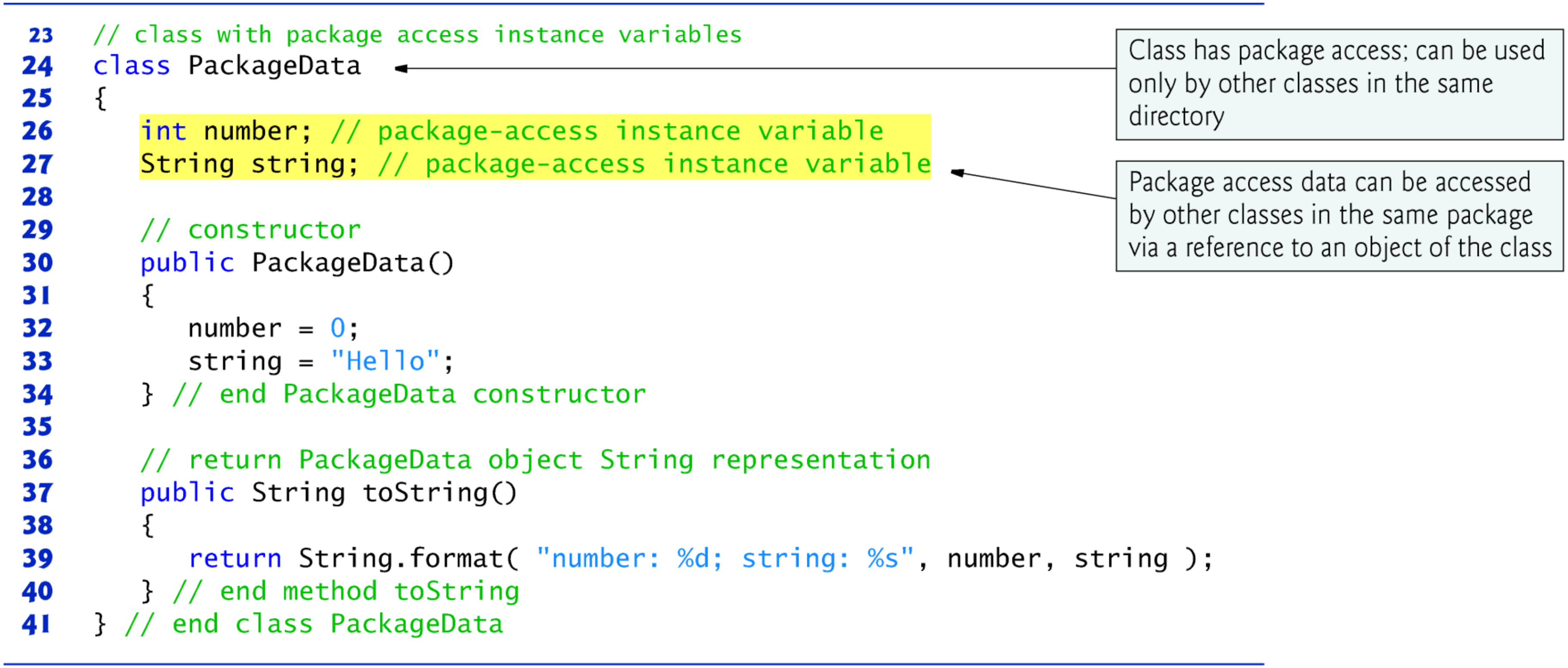
8.16 Package Access

- 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.

```java
// Fig. 8.20: PackageDataTest.java
// Package-access members of a class are accessible by other classes
// in the same package.

public class PackageDataTest
{
   public static void main( String[] args )
   {
      PackageData packageData = new PackageData();

      // output String representation of packageData
      System.out.printf( "After instantiation:\n%s\n", packageData );

      // change package access data in packageData object
      packageData.number = 77;
      packageData.string = "Goodbye";

      // output String representation of packageData
      System.out.printf( "\nAfter changing values:\n%s\n", packageData );
   } // end main
} // end class PackageDataTest

```

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.)



```java
// class with package access instance variables
class PackageData
{
   int number; // package-access instance variable
   String string; // package-access instance variable

   // constructor
   public PackageData()
   {
      number = 0;
      string = "Hello";
   } // end PackageData constructor

   // return PackageData object String representation
   public String toString()
   {
      return String.format( "number: %d; string: %s", number, string );
   } // end method toString
} // 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.)

Lab Session: static

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

End of Part I

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