



# Chapter 8

# Classes and Objects: A Deeper Look



# Objectives

- ▶ In this chapter you'll:
  - See additional details of creating class declarations.
  - Use the **throw** statement to indicate that a problem has occurred.
  - Use keyword **this in a constructor** to call another constructor in the same class.
  - Use **static** variables and methods.
  - **Import static members** of a class.
  - Use the **enum** type to create sets of constants with unique identifiers.
  - Declare enum constants with parameters.
  - **BigDecimal** for precise monetary calculations.



## 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; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime( int h, int m, int s )
13    {
14        // validate hour, minute and second
15        if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
16            ( s >= 0 && s < 60 ) )
17        {
18            hour = h;
19            minute = m;
20            second = s;
21        } // end if
```

---

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



---

```
22     else
23         throw new IllegalArgumentException(
24             "hour, minute and/or second was out of range" );
25 } // end method setTime
26
27 // convert to String in universal-time format (HH:MM:SS)
28 public String toUniversalString()
29 {
30     return String.format( "%02d:%02d:%02d", hour, minute, second );
31 } // end method toUniversalString
32
33 // convert to String in standard-time format (H:MM:SS AM or PM)
34 public String toString()
35 {
36     return String.format( "%d:%02d:%02d %s",
37             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
38             minute, second, ( hour < 12 ? "AM" : "PM" ) );
39 } // end method toString
40 } // end class Time1
```

---

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

**Fig. 8.2** | Time1 object used in an application. (Part I of 3.)



---

```
25
26     // attempt to set time with invalid values
27     try
28     {
29         time.setTime( 99, 99, 99 ); // all values out of range
30     } // end try
31     catch ( IllegalArgumentException e )
32     {
33         System.out.printf( "Exception: %s\n\n", e.getMessage() );
34     } // end catch
35
36     // display time after attempt to set invalid values
37     System.out.println( "After attempting invalid settings:" );
38     System.out.print( "Universal time: " );
39     System.out.println( time.toUniversalString() );
40     System.out.print( "Standard time: " );
41     System.out.println( time.toString() );
42 } // end main
43 } // end class Time1Test
```

---

**Fig. 8.2** | Time1 object used in an application. (Part 2 of 3.)



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

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

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

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

**Fig. 8.2** | Time1 object used in an application. (Part 3 of 3.)



## 8.2 Time Class Case Study (Cont.)

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



## 8.2 Time Class Case Study (Cont.)

- ▶ Method `setTime` and Throwing Exceptions
  - For incorrect values, `setTime` throws an exception of type **IllegalArgumentException** (lines 23–24)
    - Notifies the client code that an invalid argument was passed to the method.
    - Can use `try...catch` to catch exceptions and attempt to recover from them.
    - The `throw` statement (line 23) creates a new object of type **IllegalArgumentException**. In this case, we call the constructor that allows us to specify a custom error message.
    - After the exception object is created, the `throw` statement immediately terminates method `setTime` and the exception is thrown back to the code that attempted to set the time.



## 8.2 Time Class Case Study (Cont.)

- ▶ Java SE 8 introduced a new Date/ Time/ LocalDateTime... API—defined by the classes in the package `java.time`.
- ▶ Applications built with Java SE 8 should use the Date/Time API's capabilities.
- ▶ `LocalDateTime.now()` Obtains the current date-time from the system clock in the default time-zone.



## Software Engineering Observation 8.1

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



## Software Engineering Observation 8.2

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



## 8.3 Controlling Access to Members

- ▶ Access modifiers **public** and **private** control access to a class's variables and methods.
  - Chapter 9 introduces access modifier **protected**.
- ▶ **public** methods present to the class's clients a view of the services the class provides (the class's **public** interface).
- ▶ Clients need not be concerned with how the class accomplishes its tasks.
  - For this reason, the class's **private** variables and **private** methods (i.e., its implementation details) are not accessible to its clients.
- ▶ **private** class members are not accessible outside the class.



## Common Programming Error 8.1

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



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

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 I 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**.



---

```
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 the names
23 public SimpleTime( int hour, int minute, int second )
24 {
25     this.hour = hour; // set "this" object's hour
26     this.minute = minute; // set "this" object's minute
27     this.second = second; // set "this" object's second
28 } // end SimpleTime constructor
29
30 // use explicit and implicit "this" to call toUniversalString
31 public String buildString()
32 {
33     return String.format( "%24s: %s\n%24s: %s",
34         "this.toUniversalString()", this.toUniversalString(),
35         "toUniversalString()", toUniversalString() );
36 } // end method buildString
37
```

---

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

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



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

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



## Common Programming Error 8.2

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



## Error-Prevention Tip 8.1

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



## Performance Tip 8.1

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



## 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.
- ▶ 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:
11    // initializes each instance variable to zero
12    public Time2()
13    {
14        this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
15    } // end Time2 no-argument constructor
16
17    // Time2 constructor: hour supplied, minute and second defaulted to 0
18    public Time2( int h )
19    {
20        this( h, 0, 0 ); // invoke Time2 constructor with three arguments
21    } // end Time2 one-argument constructor
22
```

**Fig. 8.5** | Time2 class with overloaded constructors. (Part I of 6.)



```
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
```

**Fig. 8.5** | Time2 class with overloaded constructors. (Part 2 of 6.)



---

```
42 // Set Methods
43 // set a new time value using universal time;
44 // validate the data
45 public void setTime( int h, int m, int s )
46 {
47     setHour( h ); // set the hour
48     setMinute( m ); // set the minute
49     setSecond( s ); // set the second
50 } // end method setTime
51
52 // validate and set hour
53 public void setHour( int h )
54 {
55     if ( h >= 0 && h < 24 )
56         hour = h;
57     else
58         throw new IllegalArgumentException( "hour must be 0-23" );
59 } // end method setHour
60
```

---

**Fig. 8.5** | Time2 class with overloaded constructors. (Part 3 of 6.)



```
61 // validate and set minute
62 public void setMinute( int m )
63 {
64     if ( m >= 0 && m < 60 )
65         minute = m;
66     else
67         throw new IllegalArgumentException( "minute must be 0-59" );
68 } // end method setMinute
69
70 // validate and set second
71 public void setSecond( int s )
72 {
73     if ( s >= 0 && s < 60 )
74         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
75     else
76         throw new IllegalArgumentException( "second must be 0-59" );
77 } // end method setSecond
78
79 // Get Methods
80 // get hour value
81 public int getHour()
82 {
83     return hour;
84 } // end method getHour
```

**Fig. 8.5** | Time2 class with overloaded constructors. (Part 4 of 6.)



---

```
85
86     // get minute value
87     public int getMinute()
88     {
89         return minute;
90     } // end method getMinute
91
92     // get second value
93     public int getSecond()
94     {
95         return second;
96     } // end method getSecond
97
98     // convert to String in universal-time format (HH:MM:SS)
99     public String toUniversalString()
100    {
101        return String.format(
102            "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
103    } // end method toUniversalString
104
```

---

**Fig. 8.5** | Time2 class with overloaded constructors. (Part 5 of 6.)



---

```
105 // convert to String in standard-time format (H:MM:SS AM or PM)
106 public String toString()
107 {
108     return String.format( "%d:%02d:%02d %s",
109         ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
110         getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
111 } // end method toString
112 } // end class Time2
```

---

**Fig. 8.5** | Time2 class with overloaded constructors. (Part 6 of 6.)



## Common Programming Error 8.3

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



## Common Programming Error 8.4

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



## Software Engineering Observation 8.3

When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods (including those that are **private**).

```
1 public class PrivateTest {  
2  
3     private int number;  
4  
5     public static void main(String[] args) {  
6         PrivateTest testClass;  
7         testClass = new PrivateTest();  
8         testClass.number = 3;  
9         System.out.println(testClass.number);  
10    }  
11 }  
12  
13 }
```



Problems

@ Javadoc



Declaration



Console



JavaFX Preview

&lt;terminated&gt; PrivateTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_201.jdk/C



## Software Engineering Observation 8.4

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



---

```
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( t4 ); // 12:25:42
13
14     System.out.println( "Constructed with:" );
15     System.out.println( "t1: all arguments defaulted" );
16     System.out.printf( "%s\n", t1.toUniversalString() );
17     System.out.printf( "%s\n", t1.toString() );
18
19     System.out.println(
20         "t2: hour specified; minute and second defaulted" );
21     System.out.printf( "%s\n", t2.toUniversalString() );
22     System.out.printf( "%s\n", t2.toString() );
23
```

---

**Fig. 8.6** | Overloaded constructors used to initialize Time2 objects. (Part 1 of 4.)



---

```
24     System.out.println(
25         "t3: hour and minute specified; second defaulted" );
26     System.out.printf( "%s\n", t3.toUniversalString() );
27     System.out.printf( "%s\n", t3.toString() );
28
29     System.out.println( "t4: hour, minute and second specified" );
30     System.out.printf( "%s\n", t4.toUniversalString() );
31     System.out.printf( "%s\n", t4.toString() );
32
33     System.out.println( "t5: Time2 object t4 specified" );
34     System.out.printf( "%s\n", t5.toUniversalString() );
35     System.out.printf( "%s\n", t5.toString() );
36
37     // attempt to initialize t6 with invalid values
38     try
39     {
40         Time2 t6 = new Time2( 27, 74, 99 ); // invalid values
41     } // end try
```

---

**Fig. 8.6** | Overloaded constructors used to initialize `Time2` objects. (Part 2 of 4.)

```
42     catch ( IllegalArgumentException e )
43     {
44         System.out.printf( "\nException while initializing t6: %s\n",
45             e.getMessage() );
46     } // end catch
47 } // end main
48 } // end class Time2Test
```

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

**Fig. 8.6** | Overloaded constructors used to initialize `Time2` objects. (Part 3 of 4.)



```
t3: hour and minute specified; second defaulted
```

```
21:34:00
```

```
9:34:00 PM
```

```
t4: hour, minute and second specified
```

```
12:25:42
```

```
12:25:42 PM
```

```
t5: Time2 object t4 specified
```

```
12:25:42
```

```
12:25:42 PM
```

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

**Fig. 8.6** | Overloaded constructors used to initialize `Time2` objects. (Part 4 of 4.)



## 8.6 Default and No-Argument Constructors

- ▶ Every class must have at least one constructor.
- ▶ If you do not provide any constructors in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked.
- ▶ The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, `false` for `boolean` values and `null` for references).
- ▶ If your class declares constructors, the compiler will not create a default constructor.
  - In this case, you must declare a no-argument constructor if default initialization is required.
  - Like a default constructor, a no-argument constructor is invoked with empty parentheses.



## Common Programming Error 8.5

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



## Error-Prevention Tip 8.2

*Ensure that you do not include a return type in a constructor definition. Java allows other methods of the class besides its constructors to have the same name as the class and to specify return types. Such methods are not constructors and will not be called when an object of the class is instantiated.*



## 8.7 Notes on Set and Get Methods

- ▶ Classes often provide **public** methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) **private** instance variables.
- ▶ *Set* methods are also commonly called **mutator methods**, because they typically change an object's state—i.e., modify the values of instance variables.
- ▶ *Get* methods are also commonly called **accessor methods** or **query methods**.



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

- ▶ It would seem that providing *set* and *get* capabilities is essentially the same as making the instance variables **public**.
  - A **public** instance variable can be read or written by any method that has a reference to an object that contains that variable.
  - If an instance variable is declared **private**, a **public** *get* method certainly allows other methods to access it, but the *get* method can control how the client can access it.
  - A **public** *set* method can—and should—carefully scrutinize attempts to modify the variable's value to ensure valid values.
- ▶ Although *set* and *get* methods provide access to **private** data, it is restricted by the implementation of the methods.



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

- ▶ ***Validity Checking in Set Methods***
- ▶ The benefits of data integrity do not follow automatically simply because instance variables are declared **private**—you must provide validity checking.
- ▶ ***Predicate Methods***
- ▶ Another common use for accessor methods is to test whether a condition is true or false—such methods are often called **predicate methods**.
  - Example: `ArrayList`'s `isEmpty` method, which returns `true` if the `ArrayList` is empty.



## Software Engineering Observation 8.5

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



### Error-Prevention Tip 8.3

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



## 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    private static final int[] daysPerMonth = // days in each month
11        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31 };
12
13    // constructor: call checkMonth to confirm proper value for month;
14    // call checkDay to confirm proper value for day
15    public Date( int theMonth, int theDay, int theYear )
16    {
17        month = checkMonth( theMonth ); // validate month
18        year = theYear; // could validate year
19        day = checkDay( theDay ); // validate day
20    }
```

---

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



---

```
21     System.out.printf(
22         "Date object constructor for date %s\n", this );
23 } // end Date constructor
24
25 // utility method to confirm proper month value
26 private int checkMonth( int testMonth )
27 {
28     if ( testMonth > 0 && testMonth <= 12 ) // validate month
29         return testMonth;
30     else // month is invalid
31         throw new IllegalArgumentException( "month must be 1-12" );
32 } // end method checkMonth
33
34 // utility method to confirm proper day value based on month and year
35 private int checkDay( int testDay )
36 {
37     // check if day in range for month
38     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39         return testDay;
40 }
```

---

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



---

```
41     // check for leap year
42     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
43         ( year % 4 == 0 && year % 100 != 0 ) ) )
44         return testDay;
45
46     throw new IllegalArgumentException(
47         "day out-of-range for the specified month and year" );
48 } // end method checkDay
49
50 // return a String of the form month/day/year
51 public String toString()
52 {
53     return String.format( "%d/%d/%d", month, day, year );
54 } // end method toString
55 } // end class Date
```

---

**Fig. 8.7** | Date class declaration. (Part 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
```

---

**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 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 Enum Type

- ▶ 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 Enum Type (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.



## 8.9 Enum Type (Cont.)

- ▶ 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     JHTPC( "Java How to Program", "2012" ),
9     CHTPC( "C How to Program", "2007" ),
10    IW3HTPC( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTPC( "C++ How to Program", "2012" ),
12    VBHTPC( "Visual Basic 2010 How to Program", "2011" ),
13    CSHARPHTPC( "Visual C# 2010 How to Program", "2011" );
14
15     // instance fields
16     private final String title; // book title
17     private final String copyrightYear; // copyright year
18 }
```

---

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

---

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



All books:

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

Display a range of enum constants:

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

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



## 8.9 Enum Type (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.
  - <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/EnumSet.html>



## Common Programming Error 8.6

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



## 8.10 Garbage Collection

- ▶ 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.



## 8.10 Garbage Collection (Cont.)

- ▶ 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**
- ▶ 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 (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 8.6

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



## 8.11 static Class Members

- ▶ 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.



## 8.11 static Class Members (Cont.)

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



## Common Programming Error 8.7

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



## Common Programming Error 8.8

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



---

```
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
```

---

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

---

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



## Good Programming Practice 8.1

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



## 8.11 static Class Members (Cont.)

- ▶ String objects in Java are **immutable**—they cannot be modified after they are created.
  - Therefore, it's safe to have many references to one **String** object.
  - This is not normally the case for objects of most other classes in Java.
- ▶ ? **s1 = s1 + “abc”**
  - **s1** results in a **new String** object



```
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 }
```

---

**Fig. 8.13** | static member demonstration. (Part 1 of 3.)



```
23     // get names of Employees
24     System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25         e1.getFirstName(), e1.getLastName(),
26         e2.getFirstName(), e2.getLastName() );
27
28     // in this example, there is only one reference to each Employee,
29     // so the following two statements indicate that these objects
30     // are eligible for garbage collection
31     e1 = null;
32     e2 = null;
33 } // end main
34 } // end class EmployeeTest
```

**Fig. 8.13** | static member demonstration. (Part 2 of 3.)



```
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 3 of 3.)



## 8.11 static Class Members (Cont.)

- ▶ Objects become “eligible for garbage collection” when there are no more references to them in the program.
- ▶ 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;
```

- ▶ The following syntax imports all **static** members of a class:

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

- ▶ **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( "E = %f\n", E );
12        System.out.printf( "PI = %f\n", PI );
13    } // end main
14 } // end class StaticImportTest
```

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

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



## Common Programming Error 8.9

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



## 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**.



## 8.13 final Instance Variables (cont.)

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



## Software Engineering Observation 8.9

*Declaring an instance variable as final helps enforce the principle of least privilege. If an instance variable should not be modified, declare it to be final to prevent modification.*



## Common Programming Error 8.10

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



## Error-Prevention Tip 8.4

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



## Software Engineering Observation 8.10

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



## 8.14 Time Class Case Study: Creating Packages

- ▶ 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.



## 8.14 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.)

- ▶ A Java source-code file must have the following order:
  - a **package** declaration (if any),
  - **import** declarations (if any), then
  - class declarations.



```
1 // Fig. 8.15: Time1.java
2 // Time1 class declaration maintains the time in 24-hour format.
3 package com.deitel.jhttp.ch08;
4
5 public class Time1
6 {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // set a new time value using universal time; throw an
12    // exception if the hour, minute or second is invalid
13    public void setTime( int h, int m, int s )
14    {
15        // validate hour, minute and second
16        if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
17            ( s >= 0 && s < 60 ) )
18        {
19            hour = h;
20            minute = m;
21            second = s;
22        } // end if
```

**Fig. 8.15** | Packaging class Time1 for reuse. (Part 1 of 2.)



---

```
23     else
24         throw new IllegalArgumentException(
25             "hour, minute and/or second was out of range" );
26     } // end method setTime
27
28     // convert to String in universal-time format (HH:MM:SS)
29     public String toUniversalString()
30     {
31         return String.format( "%02d:%02d:%02d", hour, minute, second );
32     } // end method toUniversalString
33
34     // convert to String in standard-time format (H:MM:SS AM or PM)
35     public String toString()
36     {
37         return String.format( "%d:%02d:%02d %s",
38             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
39             minute, second, ( hour < 12 ? "AM" : "PM" ) );
40     } // end method toString
41 } // end class Time1
```

---

**Fig. 8.15** | Packaging class Time1 for reuse. (Part 2 of 2.)



## 8.14 Time Class Case Study: Creating Packages (Cont.)

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



## 8.14 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.jhttp.ch08;
```
- ▶ indicates that class **Time1** should be placed in the directory

```
com  
  deitel  
    jhttp  
      ch08
```
- ▶ The directory names in the **package** declaration specify the exact location of the classes in the package.



## 8.14 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.



## 8.14 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.jhttp.ch08.Time1`
- ▶ Can use the fully qualified name in programs, or **import** the class and use its **simple name** (the class name by itself).
- ▶ If another package contains a class by the same name, the fully qualified class names can be used to distinguish between the classes in the program and prevent a **name conflict** (also called a **name collision**).



## 8.14 Time Class Case Study: Creating Packages (Cont.)

- ▶ Fig. 8.15, 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.



---

```
1 // Fig. 8.16: Time1PackageTest.java
2 // Time1 object used in an application.
3 import com.deitel.jhttp.ch08.Time1; // import class Time1
4
5 public class Time1PackageTest
6 {
7     public static void main( String[] args )
8     {
9         // create and initialize a Time1 object
10        Time1 time = new Time1(); // invokes Time1 constructor
11
12        // output string representations of the time
13        System.out.print( "The initial universal time is: " );
14        System.out.println( time.toUniversalString() );
15        System.out.print( "The initial standard time is: " );
16        System.out.println( time.toString() );
17        System.out.println(); // output a blank line
18
```

---

**Fig. 8.16** | Time1 object used in an application. (Part 1 of 3.)



---

```
19     // change time and output updated time
20     time.setTime( 13, 27, 6 );
21     System.out.print( "Universal time after setTime is: " );
22     System.out.println( time.toUniversalString() );
23     System.out.print( "Standard time after setTime is: " );
24     System.out.println( time.toString() );
25     System.out.println(); // output a blank line
26
27     // attempt to set time with invalid values
28     try
29     {
30         time.setTime( 99, 99, 99 ); // all values out of range
31     } // end try
32     catch ( IllegalArgumentException e )
33     {
34         System.out.printf( "Exception: %s\n\n", e.getMessage() );
35     } // end catch
36
```

---

**Fig. 8.16** | Time1 object used in an application. (Part 2 of 3.)



```
37     // display time after attempt to set invalid values
38     System.out.println( "After attempting invalid settings:" );
39     System.out.print( "Universal time: " );
40     System.out.println( time.toUniversalString() );
41     System.out.print( "Standard time: " );
42     System.out.println( time.toString() );
43 } // end main
44 } // end class Time1PackageTest
```

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

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

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

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

**Fig. 8.16** | Time1 object used in an application. (Part 3 of 3.)



## Common Programming Error 8.11

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



## 8.14 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.



## 8.14 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.



## 8.14 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
  - [download.oracle.com/javase/6/docs/technotes/tools/index.html#genera](http://download.oracle.com/javase/6/docs/technotes/tools/index.html#genera)
  - The section entitled “General Information” contains information on setting the classpath for UNIX/Linux and Windows.



## 8.15 Package Access

- ▶ If no access modifier is specified for a method or variable when it's declared in a class, the method or variable is considered to have **package access**.
- ▶ Package access is **rarely** used.



---

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

---

**Fig. 8.17** | 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
```

---

**Fig. 8.17** | 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. 8.17** | Package-access members of a class are accessible by other classes in the same package. (Part 3 of 3.)



## 8.16 Jar File

### ▶ jar

- Java archive utility
- JAR file is a file format based on the popular ZIP file format and is used for aggregating many files into one.
- Bundles classes and packages
  - Classes and packages are downloaded as one unit (JAR file)

Client makes only one call to server



▶ **jar -cf test.jar TimeTest1.class com\\*.\***

- to create the JAR file
- **jar** is the *java archive utility* used to create JAR files
- the letter **c** indicates that we are creating a JAR files
- the letter **f** indicates that the next argument in the command line is the name of the JAR file to create
- **test.jar** is the name of the JAR file to create
- **TimeTest1.class com\\*.\*** actual files that will be included in the JAR file
- **com\\*.\*** call the class in the com directory should be included in the JAR file
- note : You can include selected files by specifying the path and file name for each individual file



## ▶ jar -tvf test.jar

- Produce the listing in the fig below.
- the letter **t** indicates that the table of contents for the JAR should be listed.
- the letter **v** indicates that the output should be verbose( the verbose output includes the file size in bytes and the date and time each file was created, in addition to the directory structure and file name)
- the letter **f** specifies that the next argument on the command line is the JAR file to use.

```
D:\java\08-09第二学期\Slide\examples\ch08\fig08_18_19>jar -tvf test.jar
  0 Tue Mar  3 15:53:46 CST 2009 META-INF/
  71 Tue Mar  3 15:53:46 CST 2009 META-INF/MANIFEST.MF
1038 Fri Jun 25 23:11:18 CST 2004 Time1PackageTest.class
  0 Sat Oct  9 00:44:18 CST 2004 com/deitel/
  0 Sat Oct  9 00:44:18 CST 2004 com/deitel/jhttp6/
  0 Sat Oct  9 00:44:18 CST 2004 com/deitel/jhttp6/ch08/
895 Fri Jun 25 23:10:56 CST 2004 com/deitel/jhttp6/ch08/Time1.class
```



# How to make a JAVA application – executable JAR archive (1)

1. create this manifest file (manifest.mf) with any text editor.

```
Manifest-Version: 1.0
```

```
Main-Class: TimeTest1
```

2. include the manifest file in the JAR (TimeTest1.jar) with the TimeTest1 class

```
jar cvfm test.jar manifest.mf *.class com\*.*
```

```
AC:jar -cfmv test.jar manifest.mf ./
```



3. Then you are able to start the TimeTest1.class by double-clicking on the TimeTest1.jar file (if the JRE is correctly installed) or by typing

```
java -jar test.jar
```

4. If you need to pass parameters, use the -D switch before the -jar switch and use the getProperty() method to get the value.

- JarParTest.java
- >java -D a=4 -jar test1.jar



# How to make a JAVA application – executable JAR archive(2)

- ▶ Another method:

1. Create the jar file

```
jar cf test.jar *.class
```

2. Extract the jar file, and modify the file  
“MANIFEST.MF”, and add the sentence:

```
Main-Class: Time1
```

3. compress volume file again(zip format)



# How to make a JAVA application – executable JAR archive

- ▶ Through program:
  - Package java.util.jar
  - Reference to :  
<http://study.pay500.com/6/s61001.htm>



# How to make a JAVA application – executable JAR archive(3)

- ▶ In eclipse:
  - File->Export...
  - select Java/Jar File next
  - select the resource to export and the export destination
  - next...
  - Generate the manifest file and select the class of the application entry point

**JAR Manifest Specification**

Customize the manifest file for the JAR file.



Specify the manifest:

 Generate the manifest file Save the manifest in the workspace Reuse and save the manifest in the workspace

Manifest file:

[Browse...](#) Use existing manifest from workspace

Manifest file:

[Browse...](#)

Seal contents:

 Seal the JAR[Details...](#) Seal some packages

Nothing sealed

[Details...](#)

Select the class of the application entry point:

Main class: ch08.EnumTest

[Browse...](#)[?](#)

&lt; Back

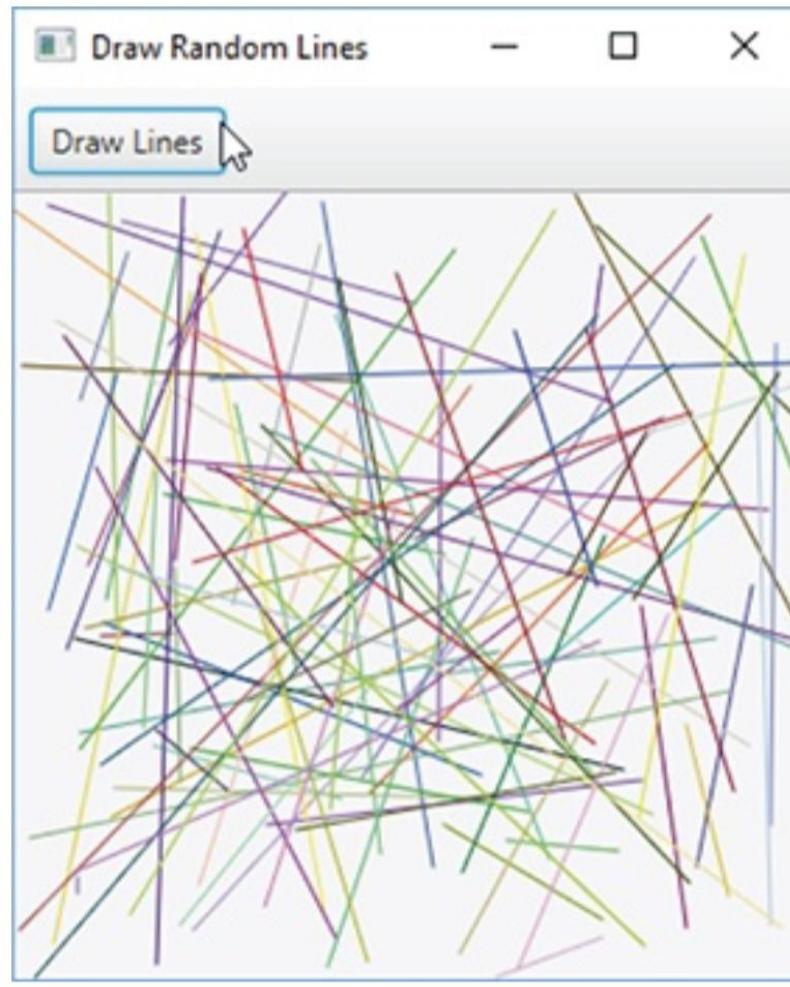
Next &gt;

Finish

Cancel



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





```
12
13 public class DrawRandomLinesController {
14     private static final SecureRandom randomNumbers = new SecureRandom();
15     @FXML private Canvas canvas;
16
17     // draws random lines
18     @FXML
19     void drawLinesButtonPressed(ActionEvent event) {
20         // get the GraphicsContext, which is used to draw on the Canvas
21         GraphicsContext gc = canvas.getGraphicsContext2D();
22
23         MyLine[] lines = new MyLine[100]; // stores the MyLine objects
24
25         final int width = (int) canvas.getWidth();
26         final int height = (int) canvas.getHeight();
27
28         // create lines
29         for (int count = 0; count < lines.length; count++) {
30             // generate random coordinates
31             int x1 = randomNumbers.nextInt(width);
32             int y1 = randomNumbers.nextInt(height);
33             int x2 = randomNumbers.nextInt(width);
34             int y2 = randomNumbers.nextInt(height);
35
36             // generate a random color
37             Color color = Color.rgb(randomNumbers.nextInt(256),
38                                     randomNumbers.nextInt(256), randomNumbers.nextInt(256));
39
40             // add a new MyLine to the array
41             lines[count] = new MyLine(x1, y1, x2, y2, color);
42         }
43
44         // clear the Canvas then draw the lines
45         gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
46
47         for (MyLine line : lines) {
48             line.draw(gc);
49         }
50     }
51 }
```



```
3+ import javafx.scene.canvas.GraphicsContext;..  
5  
6 public class MyLine {  
7     private double x1; // x-coordinate of first endpoint  
8     private double y1; // y-coordinate of first endpoint  
9     private double x2; // x-coordinate of second endpoint  
10    private double y2; // y-coordinate of second endpoint  
11    private Color strokeColor; // color of this line  
12  
13    // constructor with input values  
14@ public MyLine(  
15        double x1, double y1, double x2, double y2, Color strokeColor) {  
16  
17        this.x1 = x1;  
18        this.y1 = y1;  
19        this.x2 = x2;  
20        this.y2 = y2;  
21        this.strokeColor = strokeColor;  
22    }  
23  
24    // draw the line in the specified color  
25@ public void draw(GraphicsContext gc) {  
26        gc.setStroke(strokeColor);  
27        gc.strokeLine(x1, y1, x2, y2);  
28    }  
29 }  
30  
31
```