Chapter 8
Classes and Objects: A Deeper Look

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

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.

OBJECTIVES (cont.)

- 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.
- Use `BigDecimal` for precise monetary calculations.

OUTLINE

OUTLINE (cont.)

OUTLINE (cont.)

8.1 Introduction

- Deeper look at building classes, controlling access to members of a class and creating constructors.
- Show how to throw an exception to indicate that a problem has occurred.
- 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

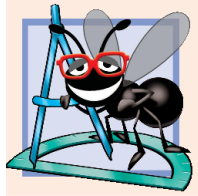
- Class `Time1` represents the time of day.
- `private int` instance variables `hour`, `minute` and `second` represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23, and minutes and seconds are each in the range 0–59).
- `public` methods `setTime`, `toUniversalString` and `toString`.
  - Clled 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; throw an
   // exception if the hour, minute or second is invalid
   public void setTime(int hour, int minute, int second) {
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
         second < 0 || second >= 60) {
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
      }

      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }
```

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

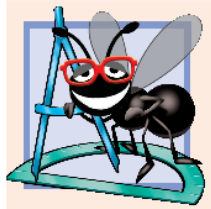
```
23
24      // convert to String in universal-time format (HH:MM:SS)
25      public String toUniversalString() {
26         return String.format("%02d:%02d:%02d", hour, minute, second);
27      }
28
29      // convert to String in standard-time format (H:MM:SS AM or PM)
30      public String toString() {
31         return String.format("%d:%02d:%02d %s",
32            ((hour == 0 || hour == 12) ? 12 : hour % 12),
33            minute, second, (hour < 12 ? "AM" : "PM"));
34      }
35   }
```

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



Software Engineering Observation 8.1

For a method like `setTime` in Fig. 8.1, validate all of the method's arguments before using them to set instance variable values to ensure that the object's data is modified only if all the arguments are valid.



Software Engineering Observation 8.2

Recall from Chapter 3 that methods declared with access modifier `private` can be called only by other methods of the class in which the `private` methods are declared. Such methods are commonly referred to as **utility methods** or **helper methods** because they're typically used to support the operation of the class's other methods.

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an app.
3
4  public class Time1Test {
5     public static void main(String[] args) {
6        // create and initialize a Time1 object
7        Time1 time = new Time1(); // invokes Time1 constructor
8
9        // output string representations of the time
10       displayTime("After time object is created", time);
11       System.out.println();
12
13       // change time and output updated time
14       time.setTime(13, 27, 6);
15       displayTime("After calling setTime", time);
16       System.out.println();
17
```

Fig. 8.2 | `Time1` object used in an app. (Part 1 of 3.)

```
18          // attempt to set time with invalid values
19          try {
20             time.setTime(99, 99, 99); // all values out of range
21          }
22          catch (IllegalArgumentException e) {
23             System.out.printf("Exception: %s%n%n", e.getMessage());
24          }
25
26          // display time after attempt to set invalid values
27          displayTime("After calling setTime with invalid values", time);
28       }
29
30       // displays a Time1 object in 24-hour and 12-hour formats
31       private static void displayTime(String header, Time1 t) {
32          System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33             header, t.toUniversalString(), t.toString());
34       }
35    }
```

Fig. 8.2 | `Time1` object used in an app. (Part 2 of 3.)

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

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

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.2 | `Time1` object used in an app. (Part 3 of 3.)

8.2 `Time` Class Case Study (Cont.)

- Class `Time1` does not declare a constructor, so the compiler supplies a default constructor.
- Each instance variable implicitly receives the default `int` value.
- 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

- Method `setTime` declares three `int` parameters and uses them to set the time.
- Lines 13–14 test each argument to determine whether the value is outside the proper range.

8.2 `Time` Class Case Study (Cont.)

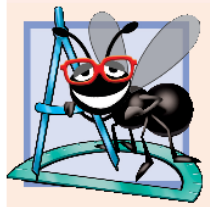
Method `setTime` and Throwing Exceptions (cont.)

- For incorrect values, `setTime` throws an exception of type `IllegalArgumentException`
  - 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 class instance creation expression in the `throw` statement 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 returned to the calling method that attempted to set the time.

8.2 `Time` Class Case Study (Cont.)

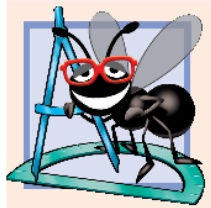
Software Engineering of the `Time1` Class Declaration

- The instance variables `hour`, `minute` and `second` are each declared `private`.
- The actual data representation used within the class is of no concern to the class's clients.
- Reasonable for `Time1` to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight.
- Clients could use the same `public` methods and get the same results without being aware of this.



Software Engineering Observation 8.3

Classes simplify programming, because the client can use only a class's `public` methods. 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 8.4

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 a class's implementation details.

8.2 `Time` Class Case Study (Cont.)

Java SE 8—Date/Time API

- Rather than building your own date and time classes, you'll typically reuse the ones provided by the Java API.
- Java SE 8 introduces a new Date/Time API—defined by the classes in the package java.time—applications built with Java SE 8 should use the Date/Time API's capabilities, rather than those in earlier Java versions.
  - fixes various issues with the older classes and provides more robust, easier-to-use capabilities for manipulating dates, times, time zones, calendars and more.
- We use some Date/Time API features in Chapter 23.
- Learn more about the Date/Time API's classes at:
  - `download.java.net/jdk8/docs/api/java/time/package-summary.html`

8.3 Controlling Access to Members

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

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

```
MemberAccessTest.java:7: error: hour has private access in Time1
      time.hour = 7; // error: hour has private access in Time1
          ^
MemberAccessTest.java:8: error: minute has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
          ^
MemberAccessTest.java:9: error: 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.

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 generates a compilation error.

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 an instance 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 an instance method's body.
- Can use the `this` reference implicitly and explicitly.

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

- When you compile a `.java` file containing more than one class, the compiler produces a separate class file with the `.class` extension for every compiled class.
- When one source-code (`.java`) file contains multiple class declarations, the compiler places both class files for those classes in the same directory.
- A source-code file can contain only *one* `public` class—otherwise, a compilation error occurs.
- Non-`public` classes can be used only by other classes in the *same package*.

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest {
5     public static void main(String[] args) {
6        SimpleTime time = new SimpleTime(15, 30, 19);
7        System.out.println(time.buildString());
8     }
9  }
10
```

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

```
11 // class SimpleTime demonstrates the "this" reference
12 class SimpleTime {
13    private int hour; // 0-23
14    private int minute; // 0-59
15    private int second; // 0-59
16
17    // if the constructor uses parameter names identical to
18    // instance variable names the "this" reference is
19    // required to distinguish between the names
20    public SimpleTime(int hour, int minute, int second) {
21       this.hour = hour; // set "this" object's hour
22       this.minute = minute; // set "this" object's minute
23       this.second = second; // set "this" object's second
24    }
25
```

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

```
26      // use explicit and implicit "this" to call toUniversalString
27      public String buildString() {
28         return String.format("%24s: %s%n%24s: %s",
29            "this.toUniversalString()", this.toUniversalString(),
30            "toUniversalString()", toUniversalString());
31      }
32
33      // convert to String in universal-time format (HH:MM:SS)
34      public String toUniversalString() {
35         // "this" is not required here to access instance variables,
36         // because method does not have local variables with same
37         // names as instance variables
38         return String.format("%02d:%02d:%02d",
39            this.hour, this.minute, this.second);
40      }
41   }
```

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

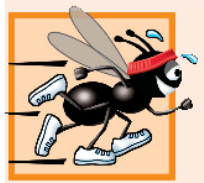
Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 3 of 3.)

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 use the `this` reference to refer to the instance variables.

Error-Prevention Tip 8.1

Most IDEs will issue a warning if you say `x = x;` instead of `this.x = x;`. The statement `x = x;` is often called a no-op (no operation).



Performance Tip 8.1

There's only one copy of each method per class—every object of the class shares the method's code. Each object, on the other hand, has its own copy of the class's instance variables. The class's non-`static` methods implicitly use `this` to determine the specific object of the class to manipulate.

8.5 `Time` Class Case Study: Overloaded Constructors

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

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

- Class `Time2` (Fig. 8.5) contains five overloaded constructors that provide convenient ways to initialize objects.
- The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration.

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

- Using `this` as shown here is a popular way to *reuse* initialization code provided by another of the class's constructors
- A constructor that calls another constructor in this manner is known as a **delegating constructor**
- Makes the class easier to maintain and modify
  - If we need to change how objects of class `Time2` are initialized, only the constructor that the class's other constructors call will need to be modified

```
1  // Fig. 8.5: Time2.java
2  // Time2 class declaration with overloaded constructors.
3
4  public class Time2 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // Time2 no-argument constructor:
10    // initializes each instance variable to zero
11    public Time2() {
12       this(0, 0, 0); // invoke constructor with three arguments
13    }
14
15    // Time2 constructor: hour supplied, minute and second defaulted to 0
16    public Time2(int hour) {
17       this(hour, 0, 0); // invoke constructor with three arguments
18    }
19
20    // Time2 constructor: hour and minute supplied, second defaulted to 0
21    public Time2(int hour, int minute) {
22       this(hour, minute, 0); // invoke constructor with three arguments
23    }
24
```

Fig. 8.5 | Time2 class declaration with overloaded constructors. (Part 1 of 6.)

```
25      // Time2 constructor: hour, minute and second supplied
26      public Time2(int hour, int minute, int second) {
27         if (hour < 0 || hour >= 24) {
28            throw new IllegalArgumentException("hour must be 0-23");
29         }
30
31         if (minute < 0 || minute >= 60) {
32            throw new IllegalArgumentException("minute must be 0-59");
33         }
34
35         if (second < 0 || second >= 60) {
36            throw new IllegalArgumentException("second must be 0-59");
37         }
38
39         this.hour = hour;
40         this.minute = minute;
41         this.second = second;
42      }
43
44      // Time2 constructor: another Time2 object supplied
45      public Time2(Time2 time) {
46         // invoke constructor with three arguments
47         this(time.hour, time.minute, time.second);
48      }
```

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

```
49
50      // Set Methods
51      // set a new time value using universal time;
52      // validate the data
53      public void setTime(int hour, int minute, int second) {
54         if (hour < 0 || hour >= 24) {
55            throw new IllegalArgumentException("hour must be 0-23");
56         }
57
58         if (minute < 0 || minute >= 60) {
59            throw new IllegalArgumentException("minute must be 0-59");
60         }
61
62         if (second < 0 || second >= 60) {
63            throw new IllegalArgumentException("second must be 0-59");
64         }
65
66         this.hour = hour;
67         this.minute = minute;
68         this.second = second;
69      }
```

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

```
70
71      // validate and set hour
72      public void setHour(int hour) {
73         if (hour < 0 || hour >= 24) {
74            throw new IllegalArgumentException("hour must be 0-23");
75         }
76
77         this.hour = hour;
78      }
79
80      // validate and set minute
81      public void setMinute(int minute) {
82         if (minute < 0 || minute >= 60) {
83            throw new IllegalArgumentException("minute must be 0-59");
84         }
85
86         this.minute = minute;
87      }
88
```

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

```
89      // validate and set second
90      public void setSecond(int second) {
91         if (second < 0 || second >= 60) {
92            throw new IllegalArgumentException("second must be 0-59");
93         }
94
95         this.second = second;
96      }
97
98      // Get Methods
99      // get hour value
100     public int getHour() {return hour;}
101
102     // get minute value
103     public int getMinute() {return minute;}
104
105     // get second value
106     public int getSecond() {return second;}
```

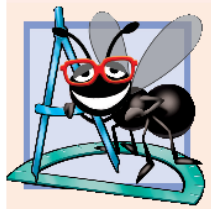
Fig. 8.5 | `Time2` class declaration with overloaded constructors. (Part 5 of 6.)

```
107
108     // convert to String in universal-time format (HH:MM:SS)
109     public String toUniversalString() {
110        return String.format(
111           "%02d:%02d:%02d", getHour(), getMinute(), getSecond());
112     }
113
114     // convert to String in standard-time format (H:MM:SS AM or PM)
115     public String toString() {
116        return String.format("%d:%02d:%02d %s",
117           ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
118           getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
119     }
120  }
```

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

Common Programming Error 8.2

It's a compilation error when `this` is used in a constructor's body to call another of the class's constructors 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`.



Software Engineering Observation 8.5

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

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test {
5     public static void main(String[] args) {
6        Time2 t1 = new Time2(); // 00:00:00
7        Time2 t2 = new Time2(2); // 02:00:00
8        Time2 t3 = new Time2(21, 34); // 21:34:00
9        Time2 t4 = new Time2(12, 25, 42); // 12:25:42
10       Time2 t5 = new Time2(t4); // 12:25:42
11
12       System.out.println("Constructed with:");
13       displayTime("t1: all default arguments", t1);
14       displayTime("t2: hour specified; default minute and second", t2);
15       displayTime("t3: hour and minute specified; default second", t3);
16       displayTime("t4: hour, minute and second specified", t4);
17       displayTime("t5: Time2 object t4 specified", t5);
```

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

```
18
19          // attempt to initialize t6 with invalid values
20          try {
21             Time2 t6 = new Time2(27, 74, 99); // invalid values
22          }
23          catch (IllegalArgumentException e) {
24             System.out.printf("%nException while initializing t6: %s%n",
25                e.getMessage());
26          }
27       }
28
29       // displays a Time2 object in 24-hour and 12-hour formats
30       private static void displayTime(String header, Time2 t) {
31          System.out.printf("%s%n   %s%n   %s%n",
32             header, t.toUniversalString(), t.toString());
33       }
34    }
```

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

```
Constructed with:
t1: all default arguments
   00:00:00
   12:00:00 AM
t2: hour specified; default minute and second
   02:00:00
   2:00:00 AM
t3: hour and minute specified; default second
   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 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.

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

Notes Regarding Class `Time2`'s* set *and* get *Methods and Constructors

- Methods can access a class's private data directly without calling the *get* methods.
- However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory).
  - If we made such a change, only the bodies of the methods that access the `private` data directly would need to change—in particular, the three-argument constructor, the `setTime` method and the individual *set* and *get* methods for the `hour`, `minute` and `second`.
  - There would be no need to modify the bodies of methods `toUniversalString` or `toString` because they do *not* access the data directly.

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

- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.
- Similarly, each `Time2` constructor could be written to include a copy of the appropriate statements from the three-argument constructor.
  - Doing so may be slightly more efficient, because the extra constructor calls are eliminated.
  - But, *duplicating* statements makes changing the class's internal data representation more difficult.
  - Having the `Time2` constructors call the constructor with three arguments requires any changes to the implementation of the three-argument constructor be made only once.

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).
- Recall that 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.

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.

Common Programming Error 8.3

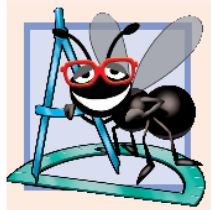
A compilation error occurs if a program attempts to initialize an object of a class by passing the wrong number or types of arguments to the class's constructor.

8.7 Notes on *Set and Get Methods*

- *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 a class's 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 at-tempts 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.



Software Engineering Observation 8.6

Classes should never have `public` nonconstant data, but declaring data `public static final` enables you to make constants available to clients of your class. For example, class `Math` offers `public static final` constants `Math.E` and `Math.PI`.

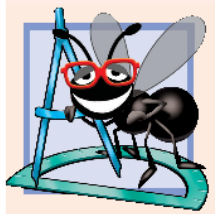
8.7 Notes on *Set and Get Methods (Cont.)*

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 and `false` otherwise.



Software Engineering Observation 8.7

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 an instance variable in an object, it's easier to debug and maintain the class. If the instance variable is not being set properly, the code that actually modifies instance variable is localized to one set method. Your debugging efforts can be focused on that one method.

Good Programming Practice 8.1

By convention, predicate method names begin with `is` rather than `get`.

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     private int month; // 1-12
6     private int day; // 1-31 based on month
7     private int year; // any year
8
9     private static final int[] daysPerMonth =
10       {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
11
```

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

```
12      // constructor: confirm proper value for month and day given the year
13      public Date(int month, int day, int year) {
14         // check if month in range
15         if (month <= 0 || month > 12) {
16            throw new IllegalArgumentException(
17               "month (" + month + ") must be 1-12");
18         }
19
20         // check if day in range for month
21         if (day <= 0 ||
22            (day > daysPerMonth[month] && !(month == 2 && day == 29))) {
23            throw new IllegalArgumentException("day (" + day +
24               ") out-of-range for the specified month and year");
25         }
26
```

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

```
27          // check for leap year if month is 2 and day is 29
28          if (month == 2 && day == 29 && !(year % 400 == 0 ||
29                (year % 4 == 0 && year % 100 != 0))) {
30             throw new IllegalArgumentException("day (" + day +
31                ") out-of-range for the specified month and year");
32          }
33
34          this.month = month;
35          this.day = day;
36          this.year = year;
37
38          System.out.printf("Date object constructor for date %s%n", this);
39       }
40
41       // return a String of the form month/day/year
42       public String toString() {
43          return String.format("%d/%d/%d", month, day, year);
44       }
45    }
```

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     private String firstName;
6     private String lastName;
7     private Date birthDate;
8     private Date hireDate;
9
10    // constructor to initialize name, birth date and hire date
11    public Employee(String firstName, String lastName, Date birthDate,
12       Date hireDate) {
13       this.firstName = firstName;
14       this.lastName = lastName;
15       this.birthDate = birthDate;
16       this.hireDate = hireDate;
17    }
18
19    // convert Employee to String format
20    public String toString() {
21       return String.format("%s, %s  Hired: %s  Birthday: %s",
22          lastName, firstName, hireDate, birthDate);
23    }
24 }
```

Fig. 8.8 | `Employee` class with references to other objects.

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest {
5     public static void main(String[] args) {
6        Date birth = new Date(7, 24, 1949);
7        Date hire = new Date(3, 12, 1988);
8        Employee employee = new Employee("Bob", "Blue", birth, hire);
9
10       System.out.println(employee);
11    }
12 }
```

```
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 Types

- 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 Types (Cont.)

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - `enum` constants are *implicitly* `final`.
  - `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.

8.9 Enum Types (Cont.)

- `enum` declarations contain two parts—the `enum` constants and the other members of the `enum` type.
- An `enum` constructor can specify any number of parameters and can be overloaded.
- For every `enum`, the compiler generates the `static` method `values` that returns an array of the `enum`'s constants.
- When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation.

```
1  // Fig. 8.10: Book.java
2  // Declaring an enum type with a constructor and explicit instance fields
3  // and accessors for these fields
4
5  public enum Book {
6     // declare constants of enum type
7     JHTP("Java How to Program", "2018"),
8     CHTP("C How to Program", "2016"),
9     IW3HTP("Internet & World Wide Web How to Program", "2012"),
10    CPPHTP("C++ How to Program", "2017"),
11    VBHTP("Visual Basic How to Program", "2014"),
12    CSHARPHTP("Visual C# How to Program", "2017");
13
14    // instance fields
15    private final String title; // book title
16    private final String copyrightYear; // copyright year
17
```

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

```
18      // enum constructor
19      Book(String title, String copyrightYear) {
20         this.title = title;
21         this.copyrightYear = copyrightYear;
22      }
23
24      // accessor for field title
25      public String getTitle() {
26         return title;
27      }
28
29      // accessor for field copyrightYear
30      public String getCopyrightYear() {
31         return copyrightYear;
32      }
33   }
```

Fig. 8.10 | Declaring an `enum` type with a 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     public static void main(String[] args) {
7        System.out.println("All books:");
8
9        // print all books in enum Book
10       for (Book book : Book.values()) {
11          System.out.printf("%-10s%-45s%s%n", book,
12             book.getTitle(), book.getCopyrightYear());
13       }
14
15       System.out.printf("%nDisplay a range of enum constants:%n");
16
17       // print first four books
18       for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTP)) {
19          System.out.printf("%-10s%-45s%s%n", book,
20             book.getTitle(), book.getCopyrightYear());
21       }
22    }
23 }
```

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

```
All books:
JHTP      Java How to Program                           2018
CHTP      C How to Program                              2016
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2017
VBHTP     Visual Basic How to Program                   2014
CSHARPHTP Visual C# How to Program                      2017

Display a range of enum constants:
JHTP      Java How to Program                           2018
CHTP      C How to Program                              2016
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2017
```

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

8.9 Enum Types (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.

Common Programming Error 8.4

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 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.
  - Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates.

8.10 Garbage Collection (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.
- Resource leaks other than memory leaks can also occur.
  - An app may open a file on disk to modify its contents.
  - If the app does not close the file, it must terminate before any other app can use the file.

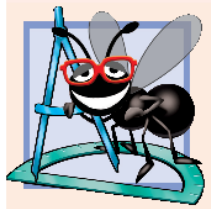
8.10 Garbage Collection (Cont.)

A Note about Class `Object`'s `finalize` Method

- Every class in Java has the methods of class `Object` (package `java.lang`), one of which is method finalize.
- You should *never* use method `finalize`, because it can cause many problems and there's uncertainty as to whether it will ever get called before a program terminates.
- The original intent of `finalize` was to allow the garbage collector to perform termination housekeeping on an object just before reclaiming the object's memory.

8.10 Garbage Collection (Cont.)

- Now, it's considered better practice for any class that uses system resources—such as files on disk—to provide a method that programmers can call to release resources when they're no longer needed in a program.
- `AutoClosable` objects reduce the likelihood of resource leaks when you use them with the try-with-resources statement.
- As its name implies, an `AutoClosable` object is closed automatically, once a try-with-resources statement finishes using the object.

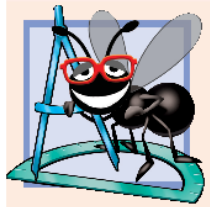


Software Engineering Observation 8.8

Many Java API classes (e.g., class `Scanner` and classes that read files from or write files to disk) provide `close` or `dispose` methods that programmers can call to release resources when they're no longer needed in a program.

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`.

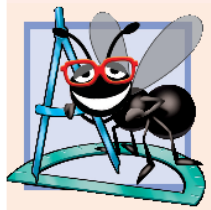


Software Engineering Observation 8.9

Use a `static` variable when all objects of a class must use the same copy of the variable.

8.11 `static` Class Members (Cont.)

- Static variables have *class scope*—they can be used in all of the class's methods.
- 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.



Software Engineering Observation 8.10

Static class variables and methods exist, and can be used, even if no objects of that class have been instantiated.

8.11 `static` Class Members (Cont.)

- A `static` method *cannot* access a class's instance variables and instance methods, 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`.

Common Programming Error 8.5

A compilation error occurs if a `static` method calls an instance 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 8.6

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     private static int count = 0; // number of Employees created
7     private String firstName;
8     private String lastName;
9
10    // initialize Employee, add 1 to static count and
11    // output String indicating that constructor was called
12    public Employee(String firstName, String lastName) {
13       this.firstName = firstName;
14       this.lastName = lastName;
15
16       ++count;  // increment static count of employees
17       System.out.printf("Employee constructor: %s %s; count = %d%n",
18          firstName, lastName, count);
19    }
```

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

```
20
21      // get first name
22      public String getFirstName() {
23         return firstName;
24      }
25
26      // get last name
27      public String getLastName() {
28         return lastName;
29      }
30
31      // static method to get static count value
32      public static int getCount() {
33         return count;
34      }
35   }
```

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 8.2

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.
- If `String` objects are immutable, you might wonder why are we able to use operators `+` and `+=` to concatenate `String` objects.
- String-concatenation actually results in a *new* `String` object containing the concatenated values—the original `String` objects are *not* modified.

```
1  // Fig. 8.13: EmployeeTest.java
2  // static member demonstration.
3
4  public class EmployeeTest {
5     public static void main(String[] args) {
6        // show that count is 0 before creating Employees
7        System.out.printf("Employees before instantiation: %d%n",
8           Employee.getCount());
9
10       // create two Employees; count should be 2
11       Employee e1 = new Employee("Susan", "Baker");
12       Employee e2 = new Employee("Bob", "Blue");
13
14       // show that count is 2 after creating two Employees
15       System.out.printf("%nEmployees after instantiation:%n");
16       System.out.printf("via e1.getCount(): %d%n", e1.getCount());
17       System.out.printf("via e2.getCount(): %d%n", e2.getCount());
18       System.out.printf("via Employee.getCount(): %d%n",
19          Employee.getCount());
```

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

```
20
21          // get names of Employees
22          System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
23             e1.getFirstName(), e1.getLastName(),
24             e2.getFirstName(), e2.getLastName());
25       }
26    }
```

```
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.)

- In a typical app, the garbage collector *might* eventually reclaim the memory for any objects that are eligible for collection.
- 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—that is, the class name and a dot (`.`) are *not* required when using an imported `static` member.
- Two forms
  - One that imports a particular `static` member (which is known as single `static` import)
  - One that imports all `static` members of a class (which is known as `static` import on demand)

8.12 `static` Import (Cont.)

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

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

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

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

- *packageName* is the package of the class and *ClassName* is the name of the class.
  - * indicates that *all* `static` members of the specified class should be available for use in the class(es) declared in the file.
- `static` import declarations import only `static` class members.
- Regular `import` statements should be used to specify the classes used in a program.

Common Programming Error 8.7

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.

```
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     public static void main(String[] args) {
7        System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
8        System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
9        System.out.printf("E = %f%n", E);
10       System.out.printf("PI = %f%n", PI);
11    }
12 }
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Fig. 8.14 | `static` import of `Math` class methods.

8.13 `final` Instance Variables

- The principle of least privilege is fundamental to good software engineering.
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.
  - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.
- Keyword `final` specifies that a variable is not modifiable (i.e., it's a constant) and any attempt to modify it is an error.

```
private final int INCREMENT;
```

  - Declares a `final` (constant) instance variable `INCREMENT` of type `int`.

8.13 `final` Instance Variables (cont.)

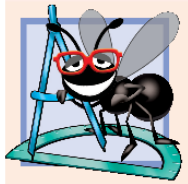
- `final` variables can be initialized when they are declared or by each of the class's constructors so that each object of the class has a different value.
- If a class provides multiple constructors, every one would be required to initialize each `final` variable.
- A `final` variable cannot be modified by assignment after it's initialized.
- If a `final` variable is not initialized, a compilation error occurs.

Common Programming Error 8.8

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

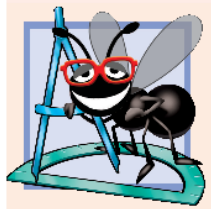
Error-Prevention Tip 8.4

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 shown that repair is often many times more expensive).



Software Engineering Observation 8.11

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. For example, in Fig. 8.8, the instance variables `firstName`, `lastName`, `birthDate` and `hireDate` are never modified after they're initialized, so they should be declared `final` (Exercise 8.20). We'll enforce this practice in all programs going forward. Testing, debugging and maintaining programs is easier when every variable that can be `final` is, in fact, `final`. You'll see additional benefits of `final` in Chapter 23, Concurrency.



Software Engineering Observation 8.12

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 Package Access

- If no access modifier is specified for a method or variable when it's declared in a class, the method or variable is considered to have package access.
- In a program uses *multiple* classes from the *same* package, these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of `static` members through the class name.
- Package access is rarely used.

```
1  // Fig. 8.15: 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     public static void main(String[] args) {
7        PackageData packageData = new PackageData();
8
9        // output String representation of packageData
10       System.out.printf("After instantiation:%n%s%n", packageData);
11
12       // change package access data in packageData object
13       packageData.number = 77;
14       packageData.string = "Goodbye";
15
16       // output String representation of packageData
17       System.out.printf("%nAfter changing values:%n%s%n", packageData);
18    }
19 }
```

Fig. 8.15 | Package-access members of a class are accessible by other classes in the same package. (Part 1 of 2.)

```
20
21   // class with package access instance variables
22   class PackageData {
23      int number = 0; // package-access instance variable
24      String string = "Hello"; // package-access instance variable
25
26      // return PackageData object String representation
27      public String toString() {
28         return String.format("number: %d; string: %s", number, string);
29      }
30   }
```

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

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

Fig. 8.15 | Package-access members of a class are accessible by other classes in the same package. (Part 2 of 2.)

8.15 Using `BigDecimal` for Precise Monetary Calculations

- In earlier chapters, we demonstrated monetary calculations using values of type `double`.
  - some `double` values are represented approximately.
- Any application that requires precise floating-point calculations—such as those in financial applications—should instead use class `BigDecimal` (from package `java.math`).

```
1  // Fig. 8.16: Interest.java
2  // Compound-interest calculations with BigDecimal.
3  import java.math.BigDecimal;
4  import java.text.NumberFormat;
5
6  public class Interest {
7     public static void main(String args[]) {
8        // initial principal amount before interest
9        BigDecimal principal = BigDecimal.valueOf(1000.0);
10       BigDecimal rate = BigDecimal.valueOf(0.05); // interest rate
11
12       // display headers
13       System.out.printf("%s%20s%n", "Year", "Amount on deposit");
14
```

Fig. 8.16 | Compound-interest calculations with `BigDecimal`. (Part 1 of 3.)

```
15          // calculate amount on deposit for each of ten years
16          for (int year = 1; year <= 10; year++) {
17             // calculate new amount for specified year
18             BigDecimal amount =
19                principal.multiply(rate.add(BigDecimal.ONE).pow(year));
20
21             // display the year and the amount
22             System.out.printf("%4d%20s%n", year,
23                NumberFormat.getCurrencyInstance().format(amount));
24          }
25       }
26    }
```

Fig. 8.16 | Compound-interest calculations with `BigDecimal`. (Part 2 of 3.)

```
Year   Amount on deposit
   1           $1,050.00
   2           $1,102.50
   3           $1,157.62
   4           $1,215.51
   5           $1,276.28
   6           $1,340.10
   7           $1,407.10
   8           $1,477.46
   9           $1,551.33
  10           $1,628.89
```

Fig. 8.16 | Compound-interest calculations with `BigDecimal`. (Part 3 of 3.)

8.15 Using `BigDecimal` for Precise Monetary Calculations (Cont.)

Interest Calculations Using `BigDecimal`

- Figure 8.16 reimplements the interest calculation example of Fig. 5.6 using objects of class `BigDecimal` to perform the calculations.
- We also introduce class `NumberFormat` (package `java.text`) for formatting numeric values as *locale-specific* `Strings`—for example, in the U.S. locale, the value 1234.56, would be formatted as `"1,234.56"`, whereas in many European locales it would be formatted as `"1.234,56"`.

8.15 Using `BigDecimal` for Precise Monetary Calculations (Cont.)

Rounding `BigDecimal` Values

- In addition to precise calculations, `BigDecimal` also gives you control over how values are rounded—by default all calculations are exact and *no* rounding occurs.
- If you do not specify how to round `BigDecimal` values and a given value cannot be represented exactly—such as the result of 1 divided by 3, which is 0.3333333…—an `ArithmeticException` occurs.
- You can specify the rounding mode for `BigDecimal` by supplying a `MathContext` object (package `java.math`) to class `BigDecimal`'s constructor when you create a `BigDecimal`. You may also provide a `MathContext` to various `BigDecimal` methods that perform calculations.

8.15 Using `BigDecimal` for Precise Monetary Calculations (Cont.)

- Class `MathContext` contains several pre-configured `MathContext` objects that you can learn about at
  - `http://docs.oracle.com/javase/7/docs/api/java/math/MathContext.html`
- By default, each pre-configured `MathContext` uses so called "bankers rounding" as explained for the `RoundingMode` constant `HALF_EVEN` at:
  - `http://docs.oracle.com/javase/7/docs/api/java/math/RoundingMode.html#HALF_EVEN`

8.15 Using `BigDecimal` for Precise Monetary Calculations (Cont.)

Scaling `BigDecimal` Values

- A `BigDecimal`'s scale is the number of digits to the right of its decimal point. If you need a `BigDecimal` rounded to a specific digit, you can call `BigDecimal` method `setScale`.
- For example, the following expression returns a `BigDecimal` with two digits to the right of the decimal point and using bankers rounding:
  - `amount.setScale(2, RoundingMode.HALF_EVEN)`

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

```
1  // Fig. 8.17: MyLine.java
2  // MyLine class represents a line.
3  import javafx.scene.canvas.GraphicsContext;
4  import javafx.scene.paint.Color;
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
```

Fig. 8.17 | `MyLine` class represents a line. (Part 1 of 2.)

```
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   }
```

Fig. 8.17 | `MyLine` class represents a line. (Part 2 of 2.)

```
1  // Fig. 8.18: DrawRandomLinesController.java
2  // Drawing random lines using MyLine objects.
3  import java.security.SecureRandom;
4  import javafx.event.ActionEvent;
5  import javafx.fxml.FXML;
6  import javafx.scene.canvas.Canvas;
7  import javafx.scene.canvas.GraphicsContext;
8  import javafx.scene.paint.Color;
9  import javafx.scene.shape.ArcType;
10
11 public class DrawRandomLinesController {
12    private static final SecureRandom randomNumbers = new SecureRandom();
13    @FXML private Canvas canvas;
14
```

Fig. 8.18 | Drawing random lines using `MyLine` objects. (Part 1 of 4.)

```
15      // draws random lines
16      @FXML
17      void drawLinesButtonPressed(ActionEvent event) {
18         // get the GraphicsContext, which is used to draw on the Canvas
19         GraphicsContext gc = canvas.getGraphicsContext2D();
20
21         MyLine[] lines = new MyLine[100]; // stores the MyLine objects
22
23         final nt width = (int) canvas.getWidth();
24         final int height = (int) canvas.getHeight();
25
```

Fig. 8.18 | Drawing random lines using `MyLine` objects. (Part 2 of 4.)

```
26          // create lines
27          for (int count = 0; count < lines.length; count++) {
28             // generate random coordinates
29             double x1 = randomNumbers.nextInt(width);
30             double y1 = randomNumbers.nextInt(height);
31             double x2 = randomNumbers.nextInt(width);
32             double y2 = randomNumbers.nextInt(height);
33
34             // generate a random color
35             Color color = Color.rgb(randomNumbers.nextInt(256),
36                randomNumbers.nextInt(256), randomNumbers.nextInt(256));
37
38             // add a new MyLine to the array
39             lines[count] = new MyLine(x1, y1, x2, y2, color);
40          }
41
42          // clear the Canvas then draw the lines
43          gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
44
45          for (MyLine line : lines) {
46             line.draw(gc);
47          }
48       }
49    }
```

Fig. 8.18 | Drawing random lines using `MyLine` objects. (Part 3 of 4.)

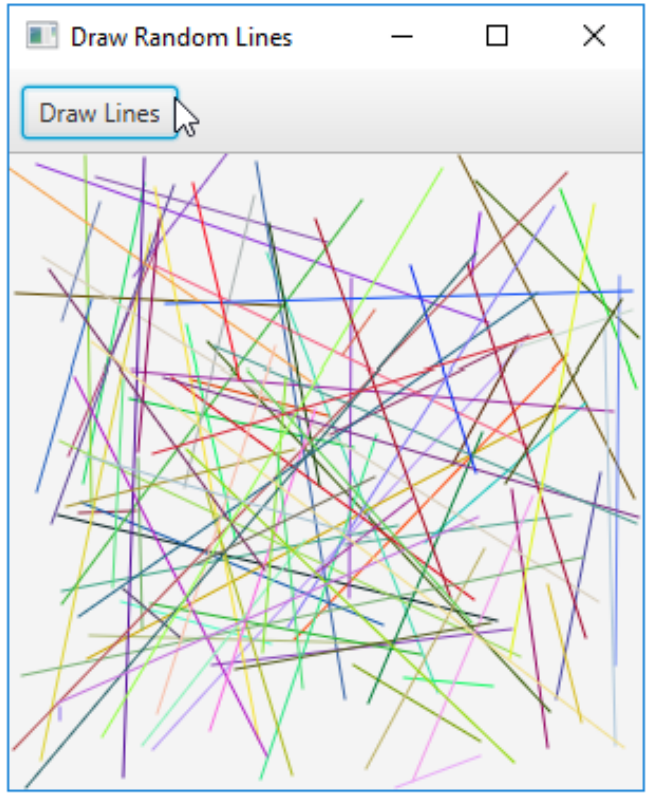


Fig. 8.18 | Drawing random lines using `MyLine` objects. (Part 4 of 4.)