Chapter 6
Methods: A Deeper Look

OBJECTIVES

In this chapter you'll learn:

- How `static` methods and fields are associated with classes rather than objects.
- How the method call/return mechanism is supported by the method-call stack.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

6.1 Introduction

- Best way to develop and maintain a large program is to construct it from small, simple pieces, or modules.
  - divide and conquer.

6.2 Program Modules in Java

- Java programs combine new methods and classes that you write with predefined methods and classes available in the Java Application Programming Interface and in other class libraries.
- Related classes are typically grouped into packages so that they can be imported into programs and reused.
  - You'll learn how to group your own classes into packages in Chapter 8.

6.2 Program Modules in Java (Cont.)

- Methods help you modularize a program by separating its tasks into self-contained units.
- Statements in method bodies
  - Written only once
  - Hidden from other methods
  - Can be reused from several locations in a program
- Divide-and-conquer approach
  - Constructing programs from small, simple pieces
- Software reusability
  - Use existing methods as building blocks to create new pro-grams.
- Dividing a program into meaningful methods makes the program easier to debug and maintain.

Software Engineering Observation 6.2

To promote software reusability, every method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.

Error-Prevention Tip 6.1

A method that performs one task is easier to test and debug than one that performs many tasks.

Software Engineering Observation 6.3

If you cannot choose a concise name that expresses a method's task, your method might be attempting to perform too many tasks. Break such a method into several smaller ones.

6.2 Program Modules in Java (Cont.)

- Hierarchical form of management (Fig. 6.1).
  - A boss (the caller) asks a worker (the called method) to perform a task and report back (return) the results after completing the task.
  - The boss method does not know how the worker method performs its designated tasks.
  - The worker may also call other worker methods, unbeknown to the boss.
- "Hiding" of implementation details promotes good software engineering.

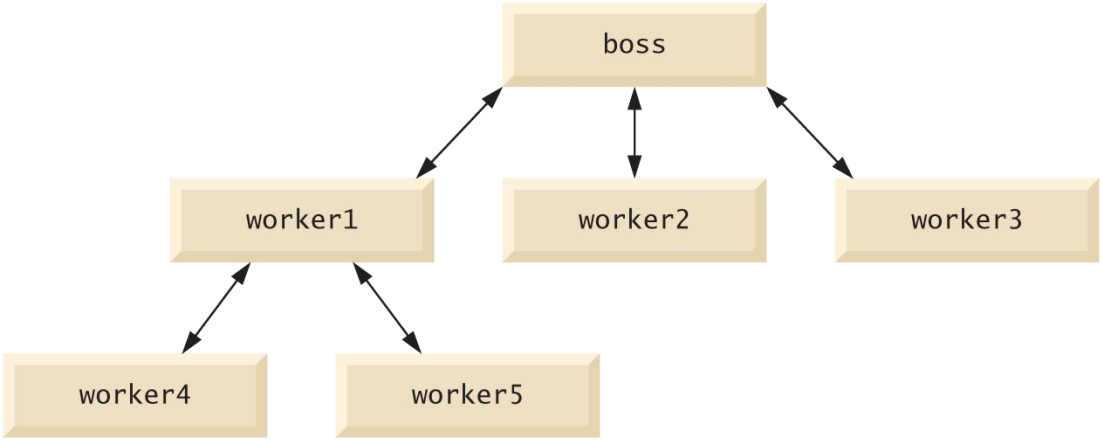

Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

6.3 `static` Methods, `static` Fields and Class `Math`

- Sometimes a method performs a task that does not depend on the contents of any object.
  - Applies to the class in which it's declared as a whole
  - Known as a `static` method or a class method
- To declare a method as `static`, place the keyword `static` before the return type in the method's declaration.
- Calling a `static` method
  - *ClassName . methodName (arguments)*
- Class `Math` provides a collection of `static` methods that enable you to perform common mathematical calculations.
- Method arguments may be constants, variables or expressions.

Software Engineering Observation 6.4

Class `Math` *is part of the* `java.lang` *package, which is implicitly imported by the compiler, so it's not necessary to import class* `Math` *to use its methods.*

Method	Description	Example

`abs( x )`	absolute value of *x*	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is `1.0` `log( Math.E * Math.E )` is `2.0`
`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`

Fig. 6.2 s. (Part 1 of 2.)

Method	Description	Example
`pow( x, y )`	x raised to the power y (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, 0.5 )` is `3.0`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is `0.0`

Fig. 6.2 | `Math` class methods. (Part 2 of 2.)

6.3 `static` Methods, `static` Fields and Class `Math` (Cont.)

- `Math` fields for common mathematical constants
  - `Math.PI` (3.141592653589793)
  - `Math.E` (2.718281828459045)
- Declared in class `Math` with the modifiers `public`, `final` and `static`
  - `public` allows you to use these fields in your own classes.
  - A field declared with keyword final is constant—its value cannot change after the field is initialized.
  - `PI` and `E` are declared `final` because their values never change.

6.3 `static` Methods, `static` Fields and Class `Math` (Cont.)

- A field that represents an attribute is also known as an instance variable—each object (instance) of the class has a separate instance of the variable in memory.
- Fields for which each object of a class does not have a separate instance of the field are declared `static` and are also known as class variables.
- All objects of a class containing `static` fields share one copy of those fields.
- Together the class variables (i.e., `static` variables) and instance variables represent the fields of a class.

6.3 `static` Methods, `static` Fields and Class `Math` (Cont.)

- Why is method **`main`** declared **`static`**?
  - The JVM attempts to invoke the `main` method of the class you specify—when no objects of the class have been created.
  - Declaring `main` as `static` allows the JVM to invoke `main` without creating an instance of the class.

6.4 Declaring Methods with Multiple Parameters

- Multiple parameters are specified as a comma-separated list.
- There must be one argument in the method call for each parameter (sometimes called a formal parameter) in the method declaration.
- Each argument must be consistent with the type of the corresponding parameter.

```
1  // Fig. 6.3: MaximumFinder.java
2  // Programmer-declared method maximum with three double parameters.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtain three floating-point values and locate the maximum value
8     public static void main( String[] args )
9     {
10       // create Scanner for input from command window
11       Scanner input = new Scanner( System.in );
12
13       // prompt for and input three floating-point values
14       System.out.print(
15          "Enter three floating-point values separated by spaces: " );
16       double number1 = input.nextDouble(); // read first double
17       double number2 = input.nextDouble(); // read second double
18       double number3 = input.nextDouble(); // read third double
19
20       // determine the maximum value
21       double result = maximum( number1, number2, number3 );
22
```

Fig. 6.3 od `maximum` with three `double` parameters.
(Part 1 of 3.)

```
23          // display maximum value
24          System.out.println( "Maximum is: " + result );
25       } // end main
26
27       // returns the maximum of its three double parameters
28       public static double maximum( double x, double y, double z )
29       {
30          double maximumValue = x; // assume x is the largest to start
31
32          // determine whether y is greater than maximumValue
33          if ( y > maximumValue )
34             maximumValue = y;
35
36          // determine whether z is greater than maximumValue
37          if ( z > maximumValue )
38             maximumValue = z;
39
40          return maximumValue;
41       } // end method maximum
42    } // end class MaximumFinder
```

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 2 of 2)

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 3 of 3.)

Software Engineering Observation 6.5

Methods can return at most one value, but the returned value could be a reference to an object that contains many values.

Software Engineering Observation 6.6

Variables should be declared as fields only if they're required for use in more than one method of the class or if the program should save their values between calls to the class's methods.

Common Programming Error 6.1

Declaring method parameters of the same type as `float x, y` *instead of* `float x, float y` *is a syntax error—a type is required for each parameter in the parameter list.*

6.4 Declaring Methods with Multiple Parameters (Cont.)

- Implementing method `maximum` by reusing method `Math.max`
  - Two calls to `Math.max`, as follows:
    - `return Math.max( x, Math.max( y, z ) );`
  - The first specifies arguments `x` and `Math.max( y, z )`.
  - Before any method can be called, its arguments must be evaluated to determine their values.
  - If an argument is a method call, the method call must be performed to determine its return value.
  - The result of the first call is passed as the second argument to the other call, which returns the larger of its two arguments.

6.4 Declaring Methods with Multiple Parameters (Cont.)

- String concatenation
  - Assemble `String` objects into larger strings with operators + or +=.
- When both operands of operator + are `String`s, operator + creates a new `String` object
- Every primitive value and object in Java has a `String` representation.
- When one of the + operator's operands is a `String`, the other is converted to a `String`, then the two are concatenated.
- If a `boolean` is concatenated with a `String`, the `boolean` is converted to the `String` `"true"` or `"false"`.
- All objects have a toString method that returns a `String` representation of the object.

Common Programming Error 6.3

Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. Java evaluates the operands of an operator from left to right. For example, if integer variable y has the value 5, the expression `"y + 2 = " + y + 2` *results in the string "y + 2 = 52", not "y + 2 = 7", because first the value of y (5) is concatenated to the string "y + 2 = ", then the value 2 is concatenated to the new larger string "y + 2 = 5". The expression* `"y + 2 = " + (y + 2)` *produces the desired result "y + 2 = 7".*

6.5 Notes on Declaring and Using Methods

- Three ways to call a method:
  - Using a method name by itself to call another method of the same class
  - Using a variable that contains a reference to an object, followed by a dot (`.`) and the method name to call a method of the referenced object
  - Using the class name and a dot (`.`) to call a `static` method of a class

6.5 Notes on Declaring and Using Methods (Cont.)

- A non-`static` method can call any method of the same class directly and can manipulate any of the class's fields directly.
- A `static` method can call *only other* `static` *methods* of the same class directly and can manipulate *only* `static` *fields* in the same class directly.
  - To access the class's non-`static` members, a `static` method must use a reference to an object of the class.

6.5 Notes on Declaring and Using Methods (Cont.)

- Three ways to return control to the statement that calls a method:
  - When the program flow reaches the method-ending right brace
  - When the following statement executes

    ```
    return;
    ```

  - When the method returns a result with a statement like

    ```
    return expression;
    ```

Common Programming Error 6.4

Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.

Common Programming Error 6.5

Placing a semicolon after the right parenthesis enclosing the parameter list of a method declaration is a syntax error.

Common Programming Error 6.6

Redeclaring a parameter as a local variable in the method's body is a compilation error.

Common Programming Error 6.7

Forgetting to return a value from a method that should return a value is a compilation error. If a return type other than `void` *is specified, the method* must *contain a* `return` *statement that returns a value consistent with the method's return type. Returning a value from a method whose return type has been declared* `void` *is a compilation error.*

6.6 Method-Call Stack and Activation Records

- Stack data structure
  - Analogous to a pile of dishes
  - A dish is placed on the pile at the top (referred to as pushing the dish onto the stack).
  - A dish is removed from the pile from the top (referred to as popping the dish off the stack).
- Last-in, first-out (LIFO) data structures
  - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

6.7 Argument Promotion and Casting

- Argument promotion
  - Converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter.
- Conversions may lead to compilation errors if Java's promotion rules are not satisfied.
- Each value is promoted to the "highest" type in the expression.
- Figure 6.4 lists the primitive types and the types to which each can be promoted.

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Fig. 6.4 | Promotions allowed for primitive types.

6.8 Java API Packages

- Java contains many predefined classes that are grouped into categories of related classes called packages.
- A great strength of Java is the Java API's thousands of classes.
- Some key Java API packages are described in Fig. 6.5.
- Java API documentation
  - https://www.cs.rit.edu/usr/local/jdk/docs/api/index.html

Package	Description
`java.applet`	The **Java Applet Package** contains a class and several interfaces required to create Java applets—programs that execute in web browsers. Applets are discussed in Chapter 23, Applets and Java Web Start; interfaces are discussed in Chapter 10, Object-Oriented Programming: Polymorphism.)
`java.awt`	The **Java Abstract Window Toolkit Package** contains the classes and interfaces required to create and manipulate GUIs in early versions of Java. In current versions, the Swing GUI components of the `javax.swing` packages are typically used instead. (Some elements of the `java.awt` package are discussed in Chapter 14, GUI Components: Part 1, Chapter 15, Graphics and Java 2D, and Chapter 25, GUI Components: Part 2.)
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)

Fig. 6.5 | J ubset). (Part 1 of 3.)

Package	Description
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 15, Graphics and Java 2D.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (See Chapter 17, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed bookwide) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See Chapter 27, Networking.)
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 28, Accessing Databases with JDBC.)
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`) and the storing and processing of large amounts of data. (See Chapter 20, Generic Collections.)

Fig. 6.5 [illegible]bset). (Part 2 of 3.)

Package	Description
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 26, Multithreading.)
`javax.media`	The **Java Media Framework Package** contains classes and interfaces for working with Java's multimedia capabilities. (See Chapter 24, Multimedia: Applets and Applications.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)
`javax.swing.event`	The **Java Swing Event Package** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)
`javax.xml.ws`	The **JAX-WS Package** contains classes and interfaces for working with web services in Java. (See Chapter 31, Web Services.)

Fig. 6.5 | Ja subset). (Part 3 of 3.)

6.9 Case Study: Secure Random-Number Generation

- Simulation and game playing
  - element of chance
  - Class SecureRandom (package `java.security`)
- Objects of class Secure-Random can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
- Difference between Random and SecureRandom
  - Random objects produce deterministic values that could be predicted by malicious programmers.
  - Secure-Random objects produce nondeterministic random numbers that cannot be predicted.

6.9 Case Study: Secure Random-Number Generation(Cont.)

- Creating a SecureRandom Object
  - SecureRandom randomNumbers = new SecureRandom();
- Obtaining a Random int Value
  - int randomValue = randomNumbers.nextInt();
- Changing the Rangle of value produced by nextInt (0,1)
  - int randomValue = randomNumbers.nextInt(2);
- For more information on the SecureRandom class, see
  - https://www.cs.rit.edu/usr/local/jdk/docs/api/java.base/java/security/SecureRandom.html

Rolling a Six-Sided Die 6,000,000 Times

Face	Frequency
1	9998153
2	10000258
3	9990950
4	10006430
5	10001190
6	10003019

```
1  package ch06.fig06_07;
2
3  // Fig. 6.7: RollDie.java
4  // Roll a six-sided die 60,000,000 times.
5  import java.security.SecureRandom;
6
7  public class RollDie {
8     public static void main(String[] args) {
9        // randomNumbers object will produce secure random numbers
10       SecureRandom randomNumbers = new SecureRandom();
11
12       int frequency1 = 0; // count of 1s rolled
13       int frequency2 = 0; // count of 2s rolled
14       int frequency3 = 0; // count of 3s rolled
15       int frequency4 = 0; // count of 4s rolled
16       int frequency5 = 0; // count of 5s rolled
17       int frequency6 = 0; // count of 6s rolled
18
```

```
18
19          // tally counts for 60,000,000 rolls of a die
20          for (int roll = 1; roll <= 60_000_000; roll++) {
21             int face = 1 + randomNumbers.nextInt(6); // number from 1 to 6
22
23             // use face value 1-6 to determine which counter to increment
24             switch (face) {
25                case 1:
26                   ++frequency1; // increment the 1s counter
27                   break;
28                case 2:
29                   ++frequency2; // increment the 2s counter
30                   break;
31                case 3:
32                   ++frequency3; // increment the 3s counter
33                   break;
34                case 4:
35                   ++frequency4; // increment the 4s counter
36                   break;
37                case 5:
38                   ++frequency5; // increment the 5s counter
39                   break;
40                case 6:
41                   ++frequency6; // increment the 6s counter
42                   break;
43             }
44          }
45
46          System.out.println("Face\tFrequency"); // output headers
47          System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
48             frequency1, frequency2, frequency3, frequency4,
49             frequency5, frequency6);
50       }
51    }
52
```

6.9.1 Generalized Scaling and Shifting of Random Numbers

- Generalize the scaling and shifting of random numbers:

```
number = shiftingValue + randomNumbers.nextInt(scalingFactor);
```

- It's also possible to choose integers at random from sets of values other than ranges of consecutive integers:

```
number = shiftingValue + differenceBetweenValues *
    randomNumbers.nextInt( scalingFactor );
```

6.10 Case Study: A Game of Chance; Introducing enum Types

- Basic rules for the dice game Craps:
  - *You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called "craps"), you lose (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

```
1  package ch06.fig06_08;
2
3  // Fig. 6.8: Craps.java
4  // Craps class simulates the dice game craps.
5  import java.security.SecureRandom;
6
7  public class Craps {
8     // create secure random number generator for use in method rollDice
9     private static final SecureRandom randomNumbers = new SecureRandom();
10
11    // enum type with constants that represent the game status
12    private enum Status {CONTINUE, WON, LOST};
13
14    // constants that represent common rolls of the dice
15    private static final int SNAKE_EYES = 2;
16    private static final int TREY = 3;
17    private static final int SEVEN = 7;
18    private static final int YO_LEVEN = 11;
19    private static final int BOX_CARS = 12;
20
21    // plays one game of craps
22    public static void main(String[] args) {
23       int myPoint = 0; // point if no win or loss on first roll
24       Status gameStatus; // can contain CONTINUE, WON or LOST
25
26       int sumOfDice = rollDice(); // first roll of the dice
27
28       // determine game status and point based on first roll
29       switch (sumOfDice) {
30          case SEVEN: // win with 7 on first roll
31          case YO_LEVEN: // win with 11 on first roll
32             gameStatus = Status.WON;
33             break;
34          case SNAKE_EYES: // lose with 2 on first roll
35          case TREY: // lose with 3 on first roll
36          case BOX_CARS: // lose with 12 on first roll
37             gameStatus = Status.LOST;
38             break;
39          default: // did not win or lose, so remember point
40             gameStatus = Status.CONTINUE; // game is not over
41             myPoint = sumOfDice; // remember the point
42             System.out.printf("Point is %d%n", myPoint);
43             break;
44       }
45
```

```
45
46          // while game is not complete
47          while (gameStatus == Status.CONTINUE) { // not WON or LOST
48             sumOfDice = rollDice(); // roll dice again
49
50             // determine game status
51             if (sumOfDice == myPoint) { // win by making point
52                gameStatus = Status.WON;
53             }
54             else {
55                if (sumOfDice == SEVEN) { // lose by rolling 7 before point
56                   gameStatus = Status.LOST;
57                }
58             }
59          }
60
61          // display won or lost message
62          if (gameStatus == Status.WON) {
63             System.out.println("Player wins");
64          }
65          else {
66             System.out.println("Player loses");
67          }
68       }
69
70       // roll dice, calculate sum and display results
71       public static int rollDice() {
72          // pick random die values
73          int die1 = 1 + randomNumbers.nextInt(6); // first die roll
74          int die2 = 1 + randomNumbers.nextInt(6); // second die roll
75
76          int sum = die1 + die2; // sum of die values
77
78          // display results of this roll
79          System.out.printf("Player rolled %d + %d = %d%n", die1, die2, sum);
80
81          return sum;
82       }
83    }
```

```
Player rolled 4 + 4 = 8
Point is 8
Player rolled 4 + 1 = 5
Player rolled 1 + 2 = 3
Player rolled 6 + 6 = 12
Player rolled 1 + 2 = 3
Player rolled 3 + 1 = 4
Player rolled 4 + 2 = 6
Player rolled 6 + 1 = 7
Player loses
```

6.10 Case Study: A Game of Chance; Introducing enum Types (Cont.)

- Notes:
- Method rollDice is declared once, but it's called from two places in main.
- **private static final** SecureRandom *randomNumbers* = **new** SecureRandom();
  - declared in class but not in methods
  - ensure us to create one SecureRandom object that's reused in each call to rollDice.

6.10 Case Study: A Game of Chance; Introducing enum Types(Cont.)

- **`enum` type `Status`**
  - An enumeration in its simplest form declares a set of constants represented by identifiers.
  - Special kind of class that is introduced by the keyword `enum` and a type name.
  - Braces delimit an `enum` declaration's body.
  - Inside the braces is a comma-separated list of enumeration constants, each representing a unique value.
  - The identifiers in an `enum` must be unique.
  - Variables of an `enum` type can be assigned only the constants declared in the enumeration.

Good Programming Practice 6.1

It's a convention to use only uppercase letters in the names of enumeration constants. This makes them stand out and reminds you that they are not variables.

Good Programming Practice 6.2

Using enumeration constants (like `Status.WON`*,* `Status.LOST` *and* `Status.CONTINUE`*) rather than literal values (such as 0, 1 and 2) makes programs easier to read and maintain.*

6.10 Case Study: A Game of Chance; Introducing enum Types(Cont.)

- Why Some Constants Are Not Defined as `enum` Constants
  - Java does not allow an `int` to be compared to an enumeration constant.
  - Java does not provide an easy way to convert an `int` value to a particular `enum` constant.
  - Translating an `int` into an `enum` constant could be done with a separate `switch` statement.
  - This would be cumbersome and not improve the readability of the program (thus defeating the purpose of using an `enum`).

6.11 Scope of Declarations

- Declarations introduce names that can be used to refer to such Java entities.
- The scope of a declaration is the portion of the program that can refer to the declared entity by its name.
  - Such an entity is said to be “in scope” for that portion of the program.

6.11 Scope of Declarations (Cont.)

- Basic scope rules:
  - The scope of a parameter declaration is the body of the method in which the declaration appears.
  - The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
  - The scope of a local-variable declaration that appears in the initialization section of a `for` statement's header is the body of the `for` statement and the other expressions in the header.
  - A method or field's scope is the entire body of the class.
- Any block may contain variable declarations.
- If a local variable or parameter in a method has the same name as a field of the class, the field is "hidden" until the block terminates execution—this is called shadowing.

Error-Prevention Tip 6.3

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method shadows a field in the class.

```
1  // Fig. 6.9: Scope.java
2  // Scope class demonstrates field and local variable scopes.
3
4  public class Scope
5  {
6     // field that is accessible to all methods of this class
7     private static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main( String[] args )
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf( "local x in main is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable has local x
18       useField(); // useField uses class Scope's field x
19       useLocalVariable(); // useLocalVariable reinitializes local x
20       useField(); // class Scope's field x retains its value
21
22       System.out.printf( "\nlocal x in main is %d\n", x );
23    } // end main
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 1 of 3.)

```
24
25      // create and initialize local variable x during each call
26      public static void useLocalVariable()
27      {
28         int x = 25; // initialized each time useLocalVariable is called
29
30         System.out.printf(
31            "\nlocal x on entering method useLocalVariable is %d\n", x );
32         ++x; // modifies this method's local variable x
33         System.out.printf(
34            "local x before exiting method useLocalVariable is %d\n", x );
35      } // end method useLocalVariable
36
37      // modify class Scope's field x during each call
38      public static void useField()
39      {
40         System.out.printf(
41            "\nfield x on entering method useField is %d\n", x );
42         x *= 10; // modifies class Scope's field x
43         System.out.printf(
44            "field x before exiting method useField is %d\n", x );
45      } // end method useField
46   } // end class Scope
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 2 of 3.)

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

Fig. 6.9 | `Scope` class demonstrates field and local variable scopes. (Part 3 of 3.)

6.12 Method Overloading

- Method overloading
  - Methods of the same name declared in the same class
  - Must have different sets of parameters
- Compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.

```
1  // Fig. 6.10: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public static void main( String[] args )
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end main
12
13    // square method with int argument
14    public static int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // end method square with int argument
20
```

Fig. 6.10 | Overloaded method declarations. (Part 1 of 2.)

```
21      // square method with double argument
22      public static double square( double doubleValue )
23      {
24         System.out.printf( "\nCalled square with double argument: %f\n",
25            doubleValue );
26         return doubleValue * doubleValue;
27      } // end method square with double argument
28   } // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.10 | Overloaded method declarations. (Part 2 of 2.)

6.12 Method Overloading (cont.)

- Distinguishing Between Overloaded Methods
  - The compiler distinguishes overloaded methods by their signatures—the methods' names and the number, types and order of their parameters.
- Return types of overloaded methods
  - *Method calls cannot be distinguished by return type.*

6.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes

- Colors displayed on computer screens are defined by their red, green, and blue components (called RGB values) that have integer values from 0 to 255.
- The higher the value of a component color, the richer that shade will be in the final color.
- Class Color (package javafx.scene.paint) represents colors using their RGB values.
- class Color contains dozens of predefined static Color objects that each can be accessed via the class name and a dot (.), as in Color.RED.

6.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes (Cont.)

```java
package ch06.GUIGraphicsCaseStudy06;

// DrawSmiley.java
// Main application class that loads and displays the DrawSmiley GUI.
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class DrawSmiley extends Application {
   @Override
   public void start(Stage stage) throws Exception {
      // loads Welcome.fxml and configures the DrawSmileyController
      Parent root =
         FXMLLoader.load(getClass().getResource("DrawSmiley.fxml"));

      Scene scene = new Scene(root); // attach scene graph to scene
      stage.setTitle("Draw Smiley"); // displayed in window's title bar
      stage.setScene(scene); // attach scene to stage
      stage.show(); // display the stage
   }

   public static void main(String[] args) {
      launch(args);
   }
}
```

```
1  package ch06.GUIGraphicsCaseStudy06;
2
3  // Fig. 3.16: DrawSmileyController.java
4  // Drawing a smiley face using colors and filled shapes.
5  import javafx.event.ActionEvent;..
10
11 public class DrawSmileyController {
12    @FXML private Canvas canvas;
13
14    // draws a smiley face
15    @FXML
16    void drawSmileyButtonPressed(ActionEvent event) {
17       // get the GraphicsContext, which is used to draw on the Canvas
18       GraphicsContext gc = canvas.getGraphicsContext2D();
19
20       // draw the face
21       gc.setFill(Color.YELLOW);
22       gc.fillOval(10, 10, 280, 280);
23       gc.strokeOval(10, 10, 280, 280);
24
25       // draw the eyes
26       gc.setFill(Color.BLACK);
27       gc.fillOval(75, 85, 40, 40);
28       gc.fillOval(185, 85, 40, 40);
29
30       // draw the mouth
31       gc.fillOval(50, 130, 200, 120);
32
33       // "touch up" the mouth into a smile
34       gc.setFill(Color.YELLOW);
35       gc.fillRect(50, 130, 200, 60);
36       gc.fillOval(50, 140, 200, 90);
37    }
38 }
39
```

6.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes(Cont.)

- Method fillOval draws a circle.
- Mehtod setStroke changes the stroke color.