Chapter 6
Methods: A Deeper Look

Java™ How to Program, 10/e

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.
- About argument promotion and casting.
- How packages group related classes.
- How to use secure 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.
- Topics in this chapter
  - `static` methods
  - Method-call stack
  - Simulation techniques with random-number generation.
  - How to declare values that cannot change (i.e., constants) in your programs.
  - Method overloading.

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 Section 21.4.10.

Software Engineering Observation 6.1

Familiarize yourself with the rich collection of classes and methods provided by the Java API (`http://docs.oracle.com/javase/7/docs/api/`*). Section 6.8 overviews several common packages. Online Appendix F explains how to navigate the API documentation. Don't reinvent the wheel. When possible, reuse Java API classes and methods. This reduces program development time and avoids introducing programming errors.*

6.2 Program Modules in Java (Cont.)

Divide and Conquer with Classes and Methods

- Classes and 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 classes and 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 Relationship Between Method Calls

- 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.

Error-Prevention Tip 6.2

When you call a method that returns a value indicating whether the method performed its task successfully, be sure to check the return value of that method and, if that method was unsuccessful, deal with the issue appropriately.

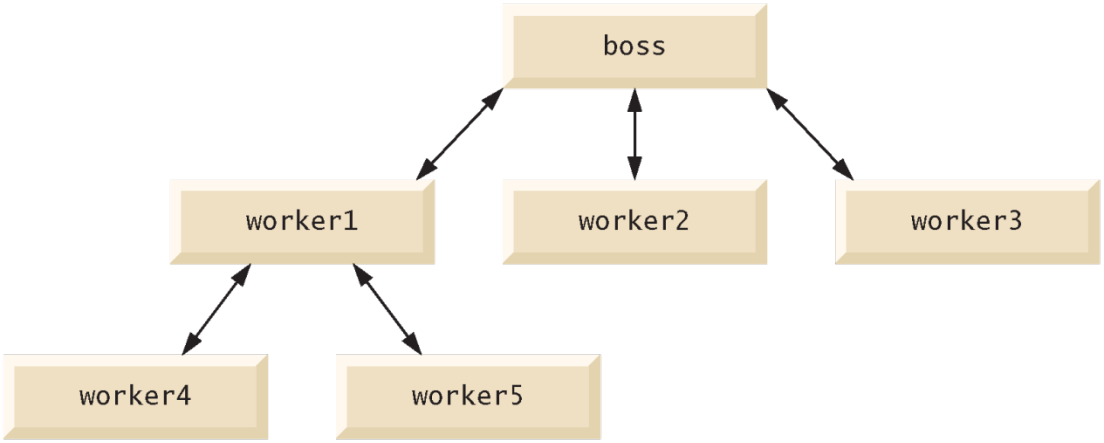

Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

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

- Sometimes a method performs a task that does not depend on an object.
  - Applies to the class in which it's declared as a whole
  - Known as a `static` method or a class method
- It's common for classes to contain convenient `static` methods to perform common tasks.
- 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*`)`

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

`Math` Class Methods

- 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 | `Math` class methods. (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.)

- Recall that each object of a class maintains its *own* copy of every instance variable of the class.
- There are variables for which each object of a class does *not* need its own separate copy (as you'll see momentarily).
- Such variables are declared static and are also known as class variables.
- When objects of a class containing `static` variables are created, all the objects of that class share one copy of those variables.
- Together a class's `static` variables and instance variables are known as its fields.
- You'll learn more about `static` fields in Section 8.11.

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

`Math` Class `static` Constants `PI` and `E`

- `Math` fields for commonly used 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.

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—at this point 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 | Programmer-declared method `maximum` with three `double` parameters. (Part 1 of 3.)

```
23          // display maximum value
24          System.out.println("Maximum is: " + result);
25       }
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       }
42    } // end class MaximumFinder
```

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 2 of 3.)

```
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.)

Assembling Strings with String Concatenation

- 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
  - characters of the right operand are placed at the end of those in the left operand
- *Every primitive value and object in Java can be represented as a `String`.*
- 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 6.2

It's a syntax error to break a `String` *literal across lines. If necessary, you can split a* `String` *into several smaller* `String`*s and use concatenation to form the desired* `String`*.*

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.)

- Non-`static` methods are typically called instance methods.
- A `static` method can call other `static` methods of the same class directly and can manipulate `static` variables in the same class directly.
  - To access the class's instance variables and instance methods, 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

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

Common Programming Error 6.6

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.*

Common Programming Error 6.7

Casting a primitive-type value to another primitive type may change the value if the new type is not a valid promotion. For example, casting a floating-point value to an integer value may introduce truncation errors (loss of the fractional part) into the result.

6.6 Method-Call Stack and Stack Frames

- 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 onto the stack is the *first* item popped from the stack.

6.6 Method-Call Stack and Activation Records (Cont.)

- When a program *calls* a method, the called method must know how to *return* to its caller
  - The *return address* of the calling method is *pushed* onto the method-call stack.
- If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order.
- The method call stack also contains the memory for the *local variables* (including the method parameters) used in each invocation of a method during a program's execution.
  - Stored as a portion of the method call stack known as the stack frame (or activation record) of the method call.

6.6 Method-Call Stack and Activation Records (Cont.)

- When a method call is made, the stack frame for that method call is *pushed* onto the method call stack.
- When the method returns to its caller, the stack frame is *popped* off the stack and those local variables are no longer known to the program.
- If more method calls occur than can have their stack frames stored on the program-execution stack, an error known as a stack overflow occurs.

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.
- Promotion rules
  - specify which conversions are allowed.
  - apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods.
- 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.

6.7 Argument Promotion and Casting (Cont.)

- Converting values to types lower in the table of Fig. 6.4 will result in different values if the lower type cannot represent the value of the higher type
- In cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator to explicitly force the conversion to occur—otherwise a compilation error occurs.

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 that we use in this book are described in Fig. 6.5.
- Overview of the packages in Java:
  - `http://docs.oracle.com/javase/7/docs/api/overview-summary.html`
  - `http://download.java.net/jdk8/docs/api/overview-summary.html`

Package	Description
`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 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 13, 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 15, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed throughout the book) 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 online Chapter 28, Networking.)
`java.security`	The **Java Security Package** contains classes and interfaces for enhancing application security.

Fig. 6.5 | Java API packages (a subset). (Part 1 of 4.)

Package	Description
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 24, Accessing Databases with JDBC.)
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable storing and processing of large amounts of data. Many of these classes and interfaces have been updated to support Java SE 8's new lambda capabilities. (See Chapter 16, Generic Collections.)
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 23, Concurrency.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. This package still uses some elements of the older `java.awt` package. (See Chapter 12, GUI Components: Part 1, and Chapter 22, 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 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)

Fig. 6.5 | Java API packages (a subset). (Part 2 of 4.)

Package	Description
`javax.xml.ws`	The JAX-WS Package contains classes and interfaces for working with web services in Java. (See online Chapter 32, REST-Based Web Services.)
`javafx` packages	JavaFX is the preferred GUI technology for the future. We discuss these packages in Chapter 25, JavaFX GUI: Part 1 and in the online JavaFX GUI and multimedia chapters.

Fig. 6.5 | Java API packages (a subset). (Part 3 of 4.)

Package	Description
Some Java SE 8 Packages Used in This Book	
`java.time`	The new Java SE 8 **Date/Time API Package** contains classes and interfaces for working with dates and times. These features are designed to replace the older date and time capabilities of package `java.util`. (See Chapter 23, Concurrency.)
`java.util.function` and `java.util.stream`	These packages contain classes and interfaces for working with Java SE 8's functional programming capabilities. (See Chapter 17, Java SE 8 Lambdas and Streams.)

Fig. 6.5 | Java API packages (a subset). (Part 4 of 4.)

6.9 Case Study: Secure Random-Number Generation

- Simulation and game playing
  - element of chance
  - Class SecureRandom (package `java.security`)
- Such objects can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
- `SecureRandom` objects produce nondeterministic random numbers that *cannot* be predicted.
- Documentation for class `SecureRandom`
  - `docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html`

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

- The range of values produced by `SecureRandom` method `nextInt` often differs from the range of values required in a particular Java application.
- `SecureRandom` method `nextInt` that receives an `int` argument returns a value from 0 up to, but not including, the argument's value.

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

Rolling a Six-Sided Die

- `face = 1 + randomNumbers.nextInt(6);`

- The argument 6—called the scaling factor—represents the number of unique values that `nextInt` should produce (0–5)
- This is called scaling the range of values
- A six-sided die has the numbers 1–6 on its faces, not 0–5.
- We shift the range of numbers produced by adding a shifting value—in this case 1—to our previous result, as in
- The shifting value (1) specifies the first value in the desired range of random integers.

```
1  // Fig. 6.6: RandomIntegers.java
2  // Shifted and scaled random integers.
3  import java.security.SecureRandom; // program uses class SecureRandom
4
5  public class RandomIntegers
6  {
7     public static void main(String[] args)
8     {
9        // randomNumbers object will produce secure random numbers
10       SecureRandom randomNumbers = new SecureRandom();
11
12       // loop 20 times
13       for (int counter = 1; counter <= 20; counter++)
14       {
15          // pick random integer from 1 to 6
16          int face = 1 + randomNumbers.nextInt(6);
17
18          System.out.printf("%d  ", face); // display generated value
19
20          // if counter is divisible by 5, start a new line of output
21          if (counter % 5 == 0)
22             System.out.println();
23       }
24    }
25 } // end class RandomIntegers
```

Fig. 6.6 | Shifted and scaled random integers. (Part 1 of 2.)

```
1   5   3   6   2
5   2   6   5   2
4   4   4   2   6
3   1   6   2   2
```

```
6   5   4   2   6
1   2   5   1   3
6   3   2   2   1
6   4   2   6   4
```

Fig. 6.6 | Shifted and scaled random integers. (Part 2 of 2.)

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

- Fig 6.7: Rolling a Six-Sided Die 6,000,000 Times

```
1  // Fig. 6.7: RollDie.java
2  // Roll a six-sided die 6,000,000 times.
3  import java.security.SecureRandom;
4
5  public class RollDie
6  {
7     public static void main(String[] args)
8     {
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
19       // tally counts for 6,000,000 rolls of a die
20       for (int roll = 1; roll <= 6000000; roll++)
21       {
22          int face = 1 + randomNumbers.nextInt(6); // number from 1 to 6
23
```

Fig. 6.7 | Roll a six-sided die 6,000,000 times. (Part 1 of 3.)

```
24            // use face value 1-6 to determine which counter to increment
25            switch (face)
26            {
27               case 1:
28                  ++frequency1; // increment the 1s counter
29                  break;
30               case 2:
31                  ++frequency2; // increment the 2s counter
32                  break;
33               case 3:
34                  ++frequency3; // increment the 3s counter
35                  break;
36               case 4:
37                  ++frequency4; // increment the 4s counter
38                  break;
39               case 5:
40                  ++frequency5; // increment the 5s counter
41                  break;
42               case 6:
43                  ++frequency6; // increment the 6s counter
44                  break;
45            }
46         }
47
```

Fig. 6.7 | Roll a six-sided die 6,000,000 times. (Part 2 of 3.)

```
48         System.out.println("Face\tFrequency"); // output headers
49         System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
50            frequency1, frequency2, frequency3, frequency4,
51            frequency5, frequency6);
52      }
53   } // end class RollDie
```

```
Face    Frequency
1       999501
2       1000412
3       998262
4       1000820
5       1002245
6       998760
```

```
Face    Frequency
1       999647
2       999557
3       999571
4       1000376
5       1000701
6       1000148
```

Fig. 6.7 | Roll a six-sided die 6,000,000 times. (Part 3 of 3.)

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  // Fig. 6.8: Craps.java
2  // Craps class simulates the dice game craps.
3  import java.security.SecureRandom;
4
5  public class Craps
6  {
7     // create secure random number generator for use in method rollDice
8     private static final SecureRandom randomNumbers = new SecureRandom();
9
10    // enum type with constants that represent the game status
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constants that represent common rolls of the dice
14    private static final int SNAKE_EYES = 2;
15    private static final int TREY = 3;
16    private static final int SEVEN = 7;
17    private static final int YO_LEVEN = 11;
18    private static final int BOX_CARS = 12;
19
```

Fig. 6.8 | `Craps` class simulates the dice game craps. (Part 1 of 5.)

```
20      // plays one game of craps
21      public static void main(String[] args)
22      {
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         {
31            case SEVEN: // win with 7 on first roll
32            case YO_LEVEN: // win with 11 on first roll
33               gameStatus = Status.WON;
34               break;
35            case SNAKE_EYES: // lose with 2 on first roll
36            case TREY: // lose with 3 on first roll
37            case BOX_CARS: // lose with 12 on first roll
38               gameStatus = Status.LOST;
39               break;
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 2 of 5.)

```
40             default: // did not win or lose, so remember point
41                gameStatus = Status.CONTINUE; // game is not over
42                myPoint = sumOfDice; // remember the point
43                System.out.printf("Point is %d%n", myPoint);
44                break;
45          }
46
47          // while game is not complete
48          while (gameStatus == Status.CONTINUE) // not WON or LOST
49          {
50             sumOfDice = rollDice(); // roll dice again
51
52             // determine game status
53             if (sumOfDice == myPoint) // win by making point
54                gameStatus = Status.WON;
55             else
56                if (sumOfDice == SEVEN) // lose by rolling 7 before point
57                   gameStatus = Status.LOST;
58          }
59
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 3 of 5.)

```
60          // display won or lost message
61          if (gameStatus == Status.WON)
62             System.out.println("Player wins");
63          else
64             System.out.println("Player loses");
65       }
66
67       // roll dice, calculate sum and display results
68       public static int rollDice()
69       {
70          // pick random die values
71          int die1 = 1 + randomNumbers.nextInt(6); // first die roll
72          int die2 = 1 + randomNumbers.nextInt(6); // second die roll
73
74          int sum = die1 + die2; // sum of die values
75
76          // display results of this roll
77          System.out.printf("Player rolled %d + %d = %d%n",
78             die1, die2, sum);
79
80          return sum;
81       }
82    } // end class Craps
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 4 of 5.)

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

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

Fig. 6.8 | `Craps` class simulates the dice game craps. (Part 5 of 5.)

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

- Notes:
  - `myPoint` is initialized to `0` to ensure that the application will compile.
  - If you do not initialize `myPoint`, the compiler issues an error, because `myPoint` is not assigned a value in every `case` of the `switch` statement, and thus the program could try to use `myPoint` before it is assigned a value.
  - `gameStatus` is assigned a value in every `case` of the `switch` statement (including the default case)—thus, it's guaranteed to be initialized before it's used, so we do not need to initialize it.

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

enum type `Status`

- An `enum` type 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 `enum` 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 `enum`.

Good Programming Practice 6.1

Use only uppercase letters in the names of `enum` *constants to make them stand out and remind you that they're not variables.*

Good Programming Practice 6.2

Using `enum` *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 Enumerations (Cont.)

Why Some Constants Are Not Defined as `enum` Constants

- Java does *not* allow an `int` to be compared to an `enum` 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 would 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.

```
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    }
24
```

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

```
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      }
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      }
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.)

Good Programming Practice 6.3

Declare variables as close to where they're first used as possible.

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.
- Literal integer values are treated as type `int`, so the method call in line 9 invokes the version of `square` that specifies an `int` parameter.
- Literal floating-point values are treated as type `double`, so the method call in line 10 invokes the version of `square` that specifies a `double` parameter.

```
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    }
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    }
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      }
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' *name* and the *number*, *types* and *order* of its parameters.
- Return types of overloaded methods
  - *Method calls cannot be distinguished by return type.*
- Figure 6.10 illustrates the errors generated when two methods have the same signature and different return types.
- Overloaded methods can have different return types if the methods have different parameter lists.
- Overloaded methods need not have the same number of parameters.

Common Programming Error 6.8

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.