Chapter 6
Methods: A Deeper Look

Java™ How to Program, 9/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.
- 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.
- Topics in this chapter
  - `static` methods
  - Declare a method with more than one parameter
  - 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 Chapter 8.

Software Engineering Observation 6.1

Familiarize yourself with the rich collection of classes and methods provided by the Java API (`download.oracle.com/javase/6/docs/api/`*). Section 6.8 presents an overview of several common packages. Appendix E 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.)

- 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 programs.
- 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.

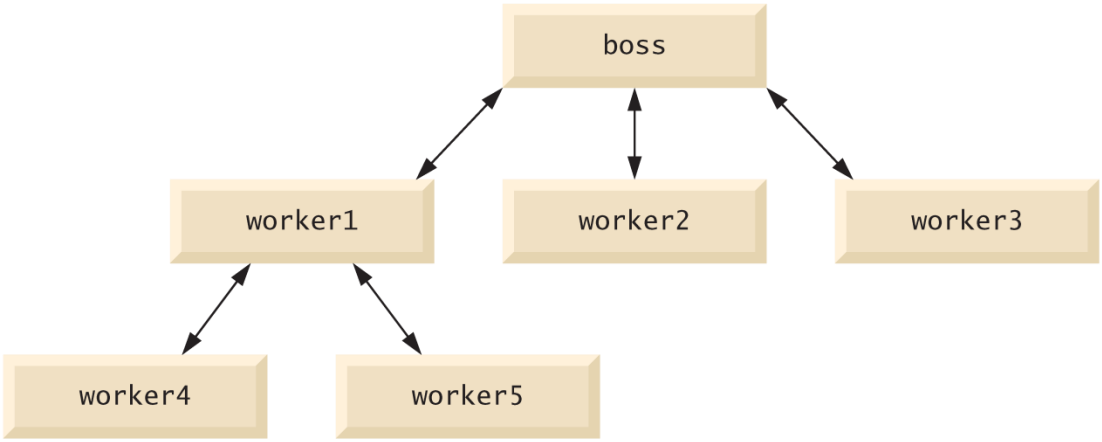

Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

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
- 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* `)`
- 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.)

- `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.

6.5 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.

6.6 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 program-execution (or 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 program-execution stack also contains the memory for the local variables used in each invocation of a method during a program's execution.
  - Stored as a portion of the program-execution stack known as the activation record or stack frame of the method call.

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

- When a method call is made, the activation record for that method call is pushed onto the program-execution stack.
- When the method returns to its caller, the method's activation record 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 activation records 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.

Common Programming Error 6.8

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

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.
- Overview of the packages in Java SE 6:
  - `download.oracle.com/javase/6/docs/api/overview-summary.html`
- Java API documentation
  - `download.oracle.com/javase/6/docs/api/`

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

6.9 Case Study: Random-Number Generation

- Simulation and game playing
  - element of chance
  - Class `Random` (package `java.util`)
  - `static` method `random` of class `Math`.
- Objects of class `Random` can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
- `Math` method `random` can produce only `double` values in the range $0.0 \leq x < 1.0$.
- Documentation for class `Random`
  - `download.oracle.com/javase/6/docs/api/java/util/Random.html`

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

- Class `Random` produces pseudorandom numbers
  - A sequence of values produced by a complex mathematical calculation.
  - The calculation uses the current time of day to seed the random-number generator.
- The range of values produced directly by `Random` method `nextInt` often differs from the range of values required in a particular Java application.
- `Random` 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.util.Random; // program uses class Random
4
5  public class RandomIntegers
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // random number generator
10       int face; // stores each random integer generated
11
12       // loop 20 times
13       for ( int counter = 1; counter <= 20; counter++ )
14       {
15          // pick random integer from 1 to 6
16          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       } // end for
24    } // end main
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.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // random number generator
10
11       int frequency1 = 0; // maintains count of 1s rolled
12       int frequency2 = 0; // count of 2s rolled
13       int frequency3 = 0; // count of 3s rolled
14       int frequency4 = 0; // count of 4s rolled
15       int frequency5 = 0; // count of 5s rolled
16       int frequency6 = 0; // count of 6s rolled
17
18       int face; // most recently rolled value
19
20       // tally counts for 6,000,000 rolls of a die
21       for ( int roll = 1; roll <= 6000000; roll++ )
22       {
23          face = 1 + randomNumbers.nextInt( 6 ); // number from 1 to 6
24
```

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

```
25            // determine roll value 1-6 and increment appropriate counter
26            switch ( face )
27            {
28               case 1:
29                  ++frequency1; // increment the 1s counter
30                  break;
31               case 2:
32                  ++frequency2; // increment the 2s counter
33                  break;
34               case 3:
35                  ++frequency3; // increment the 3s counter
36                  break;
37               case 4:
38                  ++frequency4; // increment the 4s counter
39                  break;
40               case 5:
41                  ++frequency5; // increment the 5s counter
42                  break;
43               case 6:
44                  ++frequency6; // increment the 6s counter
45                  break; // optional at end of switch
46            } // end switch
47         } // end for
48
```

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

```
49         System.out.println( "Face\tFrequency" ); // output headers
50         System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51            frequency1, frequency2, frequency3, frequency4,
52            frequency5, frequency6 );
53      } // end main
54   } // 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.)

6.9.1 Generalized Scaling and Shifting of Random Numbers

- Generalize the scaling and shifting of random numbers:

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

  where *shiftingValue* specifies the first number in the desired range of consecutive integers and *scalingFactor* specifies how many numbers are in the range.

- 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 );
```

  where *shiftingValue* specifies the first number in the desired range of values, *differenceBetweenValues* represents the constant difference between consecutive numbers in the sequence and *scalingFactor* specifies how many numbers are in the range.

6.9.2 Random-Number Repeatability for Testing and Debugging

- When debugging an application, it's sometimes useful to repeat the exact same sequence of pseudorandom numbers.
- To do so, create a `Random` object as follows:
  - ```
    Random randomNumbers =
        new Random( seedValue );
    ```
  - `seedValue` (of type `long`) seeds the random-number calculation.
- You can set a `Random` object's seed at any time during program execution by calling the object's `set` method.
    ```

Error-Prevention Tip 6.2

While developing a program, create the `Random` *object with a specific seed value to produce a repeatable sequence of numbers each time the program executes. If a logic error occurs, fix the error and test the program again with the same seed value—this allows you to reconstruct the same sequence of numbers that caused the error. Once the logic errors have been removed, create the* `Random` *object without using a seed value, causing the* `Random` *object to generate a new sequence of random numbers each time the program executes.*

6.10 Case Study: A Game of Chance; Introducing Enumerations

- 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.util.Random;
4
5  public class Craps
6  {
7     // create random number generator for use in method rollDice
8     private static final Random randomNumbers = new Random();
9
10    // enumeration 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; // optional at end of switch
45         } // end switch
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         } // end while
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       } // end main
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; // return sum of dice
81       } // end method rollDice
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 Enumerations (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—thus, it's guaranteed to be initialized before it's used and does not need to be initialized.

6.10 Case Study: A Game of Chance; Introducing Enumerations (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 Enumerations (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.
- More scope information, see the *Java Language Specification, Section 6.3: Scope of a Declaration*
  - `java.sun.com/docs/books/jls/third_edition/html/names.html#103228`

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.
- 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    } // 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.*
- 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 6.9

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

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.
- Java uses class `Color` in package `java.awt` to represent colors using their RGB values.
- Class `Color` contains 13 predefined `static Color` objects—`BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` and `YELLOW`.

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

- Class `Color` also contains a constructor of the form:
  - `public Color( int r, int g, int b )`
- so you can create custom colors by specifying the red, green and blue component values.
- `Graphics` methods `fillRect` and `fillOval` draw filled rectangles and ovals, respectively.
- `Graphics` method `setColor` sets the current drawing color.

```
1  // Fig. 6.11: DrawSmiley.java
2  // Demonstrates filled shapes.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DrawSmiley extends JPanel
8  {
9     public void paintComponent( Graphics g )
10    {
11       super.paintComponent( g );
12
13       // draw the face
14       g.setColor( Color.YELLOW );
15       g.fillOval( 10, 10, 200, 200 );
16
17       // draw the eyes
18       g.setColor( Color.BLACK );
19       g.fillOval( 55, 65, 30, 30 );
20       g.fillOval( 135, 65, 30, 30 );
21
22       // draw the mouth
23       g.fillOval( 50, 110, 120, 60 );
24
```

Fig. 6.11 | Drawing a smiley face using colors and filled shapes. (Part 1 of 2.)

```
25          // "touch up" the mouth into a smile
26          g.setColor( Color.YELLOW );
27          g.fillRect( 50, 110, 120, 30 );
28          g.fillOval( 50, 120, 120, 40 );
29       } // end method paintComponent
30    } // end class DrawSmiley
```

Fig. 6.11 | Drawing a smiley face using colors and filled shapes. (Part 2 of 2.)

```
1  // Fig. 6.12: DrawSmileyTest.java
2  // Test application that displays a smiley face.
3  import javax.swing.JFrame;
4
5  public class DrawSmileyTest
6  {
7     public static void main( String[] args )
8     {
9        DrawSmiley panel = new DrawSmiley();
10       JFrame application = new JFrame();
11
12       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       application.add( panel );
14       application.setSize( 230, 250 );
15       application.setVisible( true );
16    } // end main
17 } // end class DrawSmileyTest
```

Fig. 6.12 | Creating `JFrame` to display a smiley face. (Part 1 of 2.)

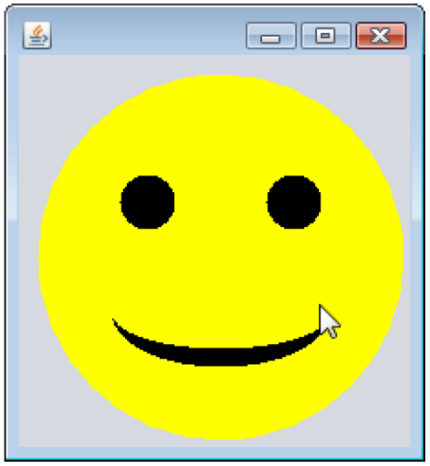

Fig. 6.12 | Creating `JFrame` to display a smiley face. (Part 2 of 2.)

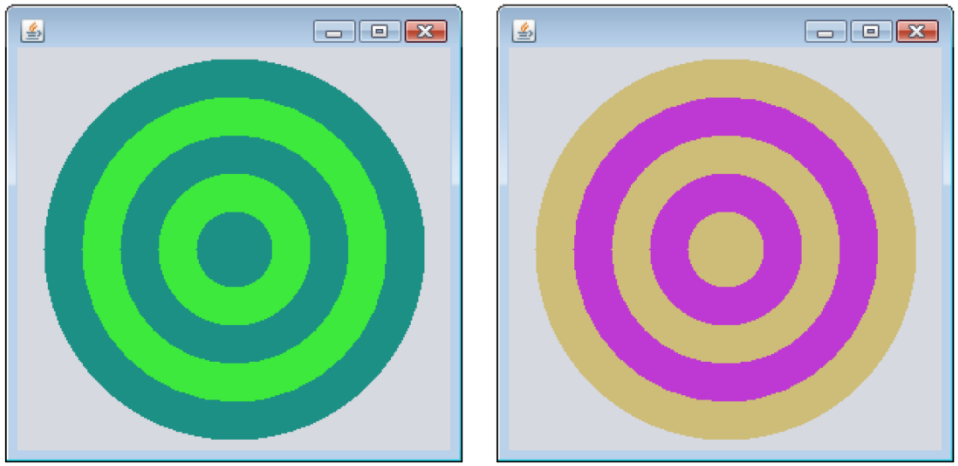

Fig. 6.13 | A bull's-eye with two alternating, random colors.

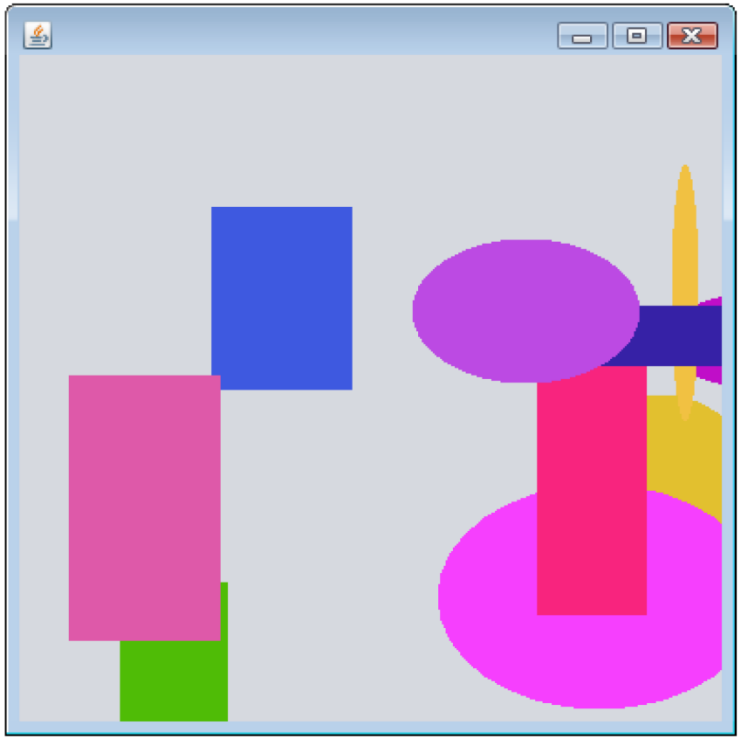

Fig. 6.14 | Randomly generated shapes.