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.

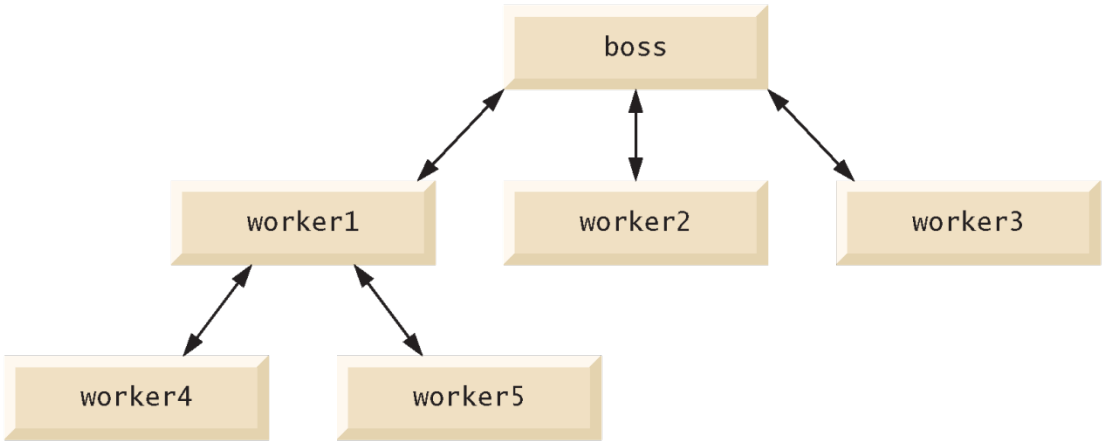

Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

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.