Methods: A Deeper Look

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.

Program Modules in Java (Cont.)

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

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.

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.

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.

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.

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.

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.


```java
// Fig. 6.3: MaximumFinder.java
// Programmer-declared method maximum with three double parameters.
import java.util.Scanner;

public class MaximumFinder
{
   // obtain three floating-point values and locate the maximum value
   public void determineMaximum()
   {
      // create Scanner for input from command window
      Scanner input = new Scanner( System.in );

      // prompt for and input three floating-point values
      System.out.print(
         "Enter three floating-point values separated by spaces: " );
      double number1 = input.nextDouble(); // read first double
      double number2 = input.nextDouble(); // read second double
      double number3 = input.nextDouble(); // read third double

      // determine the maximum value
      double result = maximum( number1, number2, number3 );
```

Passing three arguments to method `maximum`

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 1 of 2.)

```java
      // display maximum value
      System.out.println( "Maximum is: " + result );
   } // end method determineMaximum

   // returns the maximum of its three double parameters
   public double maximum( double x, double y, double z )
   {
      double maximumValue = x; // assume x is the largest to start

      // determine whether y is greater than maximumValue
      if ( y > maximumValue )
         maximumValue = y;

      // determine whether z is greater than maximumValue
      if ( z > maximumValue )
         maximumValue = z;

      return maximumValue;
   } // end method maximum
} // end class MaximumFinder
```

Method `maximum` receives three parameters and returns the largest of the three

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


```java
// Fig. 6.4: MaximumFinderTest.java
// Application to test class MaximumFinder.

public class MaximumFinderTest
{
   // application starting point
   public static void main( String[] args )
   {
      MaximumFinder maximumFinder = new MaximumFinder();
      maximumFinder.determineMaximum();
   } // end main
} // end class MaximumFinderTest
```

```
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. 6.4 | Application to test class `MaximumFinder`.

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.

Declaring Methods with Multiple Parameters (Cont.)

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

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

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.

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;
    ```


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.

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.

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.

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 6.5 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. 6.5 | Promotions allowed for primitive types.

Argument Promotion and Casting (Cont.)

- Converting values to types lower in the table of Fig. 6.5 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.

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. 6.6.

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 of Java, 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. 6.6 | Java API packages (a subset). (Part 1 of 4.)

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.)

Fig. 6.6 | Java API packages (a subset). (Part 2 of 4.)

Package	Description
`java.text`	The Java Text Package contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to locales (e.g., a program may display strings in different languages, based on the user's country).
`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.)
`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.)

Fig. 6.6 | Java API packages (a subset). (Part 3 of 4.)

Package	Description
`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. 6.6 | Java API packages (a subset). (Part 4 of 4.)

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 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.

```java
// Fig. 6.11: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private int x = 1;

   // method begin creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public void begin()
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in method begin is %d\n", x );

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf( "\nlocal x in method begin is %d\n", x );
   } // end method begin
```

Class scope

Method scope

Fig. 6.11 | Scope class demonstrating scopes of a field and local variables. (Part 1 of 2.)


```java

   // create and initialize local variable x during each call
   public void useLocalVariable()
   {
      int x = 25; // initialized each time useLocalVariable is called

      System.out.printf(
         "\nlocal x on entering method useLocalVariable is %d\n", x );
      ++x; // modifies this method's local variable x
      System.out.printf(
         "local x before exiting method useLocalVariable is %d\n", x );
   } // end method useLocalVariable

   // modify class Scope's field x during each call
   public void useField()
   {
      System.out.printf(
         "\nfield x on entering method useField is %d\n", x );
      x *= 10; // modifies class Scope's field x
      System.out.printf(
         "field x before exiting method useField is %d\n", x );
   } // end method useField
} // end class Scope
```

Method scope

Uses instance variable x

Fig. 6.11 | Scope class demonstrating scopes of a field and local variables. (Part 2 of 2.)

```java
// Fig. 6.12: ScopeTest.java
// Application to test class Scope.

public class ScopeTest
{
   // application starting point
   public static void main( String[] args )
   {
      Scope testScope = new Scope();
      testScope.begin();
   } // end main
} // end class ScopeTest
```

Fig. 6.12 | Application to test class Scope. (Part 1 of 2.)


```
local x in method begin 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 method begin is 5
```

Fig. 6.12 | Application to test class Scope. (Part 2 of 2.)

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. 6.13: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public void testOverloadedMethods()
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 method testOverloadedMethods
12
13    // square method with int argument
14    public 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
```

Calls `square` with an `int` parameter

Calls `square` with a `double` parameter

`square` method that receives an `int`

Fig. 6.13 | Overloaded method declarations. (Part 1 of 2.)

```java
    // square method with double argument
    public double square( double doubleValue )
    {
       System.out.printf( "\nCalled square with double argument: %f\n",
          doubleValue );
       return doubleValue * doubleValue;
    } // end method square with double argument
 } // end class MethodOverload
```

square method that receives a double

Fig. 6.13 | Overloaded method declarations. (Part 2 of 2.)


```java
 // Fig. 6.14: MethodOverloadTest.java
 // Application to test class MethodOverload.

 public class MethodOverloadTest
 {
    public static void main( String[] args )
    {
       MethodOverload methodOverload = new MethodOverload();
       methodOverload.testOverloadedMethods();
    } // end main
 } // end class MethodOverloadTest
```

```
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. 6.14 | Application to test class `MethodOverload`.

Method Overloading

- 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 6.15 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.


```java
// Fig. 6.15: MethodOverloadError.java
// Overloaded methods with identical signatures
// cause compilation errors, even if return types are different.

public class MethodOverloadError
{
   // declaration of method square with int argument
   public int square( int x )
   {
      return x * x;
   }

   // second declaration of method square with int argument
   // causes compilation error even though return types are different
   public double square( int y )
   {
      return y * y;
   }
} // end class MethodOverloadError
```

Generates a compilation error

Fig. 6.15 | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different. (Part 1 of 2.)

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Fig. 6.15 | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different. (Part 2 of 2.)