University of Babylon Assistant Lecturer : Wadhah
Collage of Computer R. Baiee

Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called divide and conquer. Methods, which we introduced in Last lectures, help you modularize programs. Here , we study methods in more depth. We emphasize how to declare and use methods to facilitate the design, implementation, operation and maintenance of large programs.

Methods (called functions or procedures in some languages) help you modularize a program by separating its tasks into self-contained units. You've declared methods in every program you've written. The statements in the method bodies are written only once, are hidden from other methods and can be reused from several locations in a program. One motivation for modularizing a program into methods is the ***divide-and-conquer*** approach, which makes program development more manageable by constructing programs from small, simple pieces. Another is ***software reusability*** sing existing methods as building blocks to create new programs. Often, you can create programs mostly from standardized methods rather than by building customized code. For example, in earlier programs, we did not define how to read data from the keyboard Java provides these capabilities in the methods of class Scanner. A third motivation is to ***avoid repeating code***. Dividing a program into meaningful methods makes the program easier to debug and maintain.

Static methods and Math Class:

We use various Math class methods here to present the concept of static methods. Class Math provides a collection of methods that enable you to perform common mathematical calculations. For example, you can calculate the square root of 900.0 with the static method call.

```
Math.sqrt( 900.0 )
```

Exercise :

Build your own ***MyMath*** class with many common mathematical methods without using default Java Math class and its methods.

Declaring Methods with Multiple Parameters

Methods often require more than one piece of information to perform their tasks. We now consider how to write your own methods with multiple parameters. Figure 6.3 uses a method called maximum to determine and return the largest of three double values. In main, lines 14–18 prompt the user to enter three double values, then read them from the user. Line 21 calls method maximum (declared in lines 28–41) to determine the largest of the three values it receives as arguments. When method maximum returns the result to line 21, the program assigns maximum's return value to local variable result. Then line 24 outputs the maximum value.

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

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

Notes :

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

Implementing Method maximum by Reusing Method Math.max

The entire body of our maximum method could also be implemented with two calls to `Math.max`, as follows:

```
return Math.max( x, Math.max( y, z ) );
```

Notes on Declaring and Using Methods

There are three ways to call a method:

1. Using a method name by itself to call another method of the *same* class such as `maximum(number1, number2, number3)` in line 21 of Fig. 6.3.
2. Using a variable that contains a reference to an object, followed by a dot (.) and the method name to call a non-static method of the referenced object such as, **myGradeBook.displayMessage(),** which calls a method of class `GradeBook` from the `main` method of `GradeBookTest`.
3. Using the class name and a dot (.) to call a static method of a class such as `Math.sqrt(900.0)` .

A static method can call only other static methods of the same class directly (i.e., using the method name by itself) and can manipulate only static variables 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. Recall that static methods relate to a class as a whole, whereas non-static methods are associated with a specific instance (object) of the class and may manipulate the instance variables of that object. Many objects of a class, each with its own copies of the instance variables, may exist at the same time. Suppose a static method were to invoke a non-static method directly. How would the method know which object's instance variables to manipulate? What would happen if no objects of the class existed at the time the non-static method was invoked? Thus, Java does not allow a static method to access non-static members of the same class directly.

Method-Call Stack and Activation Records

To understand how Java performs method calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack. You can think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's normally placed at the top (referred to as pushing the dish onto the stack). Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as popping the dish off the stack). Stacks are known as last-in, first-out (LIFO) data

structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

When a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the program-execution stack (sometimes referred to as 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 so that each method can return to its caller.

The program-execution stack also contains the memory for the local variables used in each invocation of a method during a program's execution. This data, stored as a portion of the program-execution stack, is known as the activation record or stack frame of the method call. 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 activation record for this method call is popped off the stack and those local variables are no longer known to the program.

Argument Promotion and Casting

Another important feature of method calls is **argument promotion**—converting an argument's value, if possible, to the type that themethod expects to receive in its corresponding parameter. For example, a program can call Math method sqrt with an int argument even though a double argument is expected. The statement

```
System.out.println( Math.sqrt( 4 ) );
```

correctly evaluates Math.sqrt(4) and prints the value 2.0. Themethod declaration's parameter list causes Java to convert the int value 4 to the double value 4.0 before passing the value to method sqrt. Such conversions may lead to compilation errors if Java's **promotion rules** are not satisfied. These rules specify which conversions are allowed—that is, which ones can be performed without losing data. In the sqrt example above, an int is converted to a double without changing its value. However, converting a double to an int truncates the fractional part of the double value—thus, part of the value is lost. Converting large integer types to small integer types (e.g., long to int, or int to short) may also result in changed values.

The promotion rules 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. Actually, the expression uses a temporary copy of each value—the types of the original values remain unchanged. Figure 6.4 lists the primitive types and the types to which each can be

promoted. The valid promotions for a given type are always to a type higher in the table. For example, an int can be promoted to the higher types long, float and double.

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 (e.g., the int value 2000000 cannot be represented as a short, and any floating-point number with digits after its decimal point cannot be represented in an integer type such as long, int or short).

Therefore, in cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator (introduced in Section 4.9) to explicitly force the conversion to occur—otherwise a compilation error occurs. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might cause loss of information, but for my purposes here, that's fine." Suppose method square calculates the square of an integer and thus requires an int argument. To call square with a double argument named doubleValue, we would be required to write the method call as

```
square( (int) doubleValue )
```

This method call explicitly casts (converts) a copy of variable doubleValue's value to an integer for use in method square. Thus, if doubleValue's value is 4.5, the method receives the value 4 and returns 16, not 20.25.

Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters)—this is called **method overloading**. When an overloaded method is called, the compiler selects the appropriate method by examining the number, types and order of the arguments in the call. Method overloading is commonly 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.

Declaring Overloaded Methods

Class MethodOverload (Fig. 6.10) includes two overloaded versions of method square— one that calculates the square of an int (and returns an int) and one that calculates the square of a double (and returns a double). Although these methods have the same name and similar parameter lists and bodies, think of them simply as *different*

methods. It may help to think of the method names as “square of int” and“square of double,” respectively

```
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
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.)

The compiler distinguishes overloaded methods by their signature—a combination of the method’s name and the number, types and order of its parameters. If the compiler looked only at method names during compilation, the code in Fig. 6.10 would be ambiguous— the compiler would not know how to distinguish between the two square methods (lines 14–19 and 22–27). Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are unique in that class.

In discussing the logical names of methods used by the compiler, we did not mention the return types of the methods. *Method calls cannot be distinguished by return type*. If you had overloaded methods that differed only by their return types and you called one of the methods in a standalone statement as in:

```
square( 2 );
```

the compiler would *not* be able to determine the version of the method to call, because the return value is ignored. When two methods have the same signature and different return types, the compiler issues an error message indicating that the method is already defined in the class. Overloaded methods *can* have different return types if the methods have different parameter lists. Also, overloaded methods need *not* have the same number of parameters.