11. Methods

Java

Summer 2008

Instructor: Dr. Masoud Yaghini

Outline

- Creating a Method
- Calling a Method
- Passing Parameters
- Overloading Methods
- The Scope of Local Variables
- Method Abstraction
- References

Creating a Method

Creating a Method

- A method is a collection of statements that are grouped together to perform an operation.

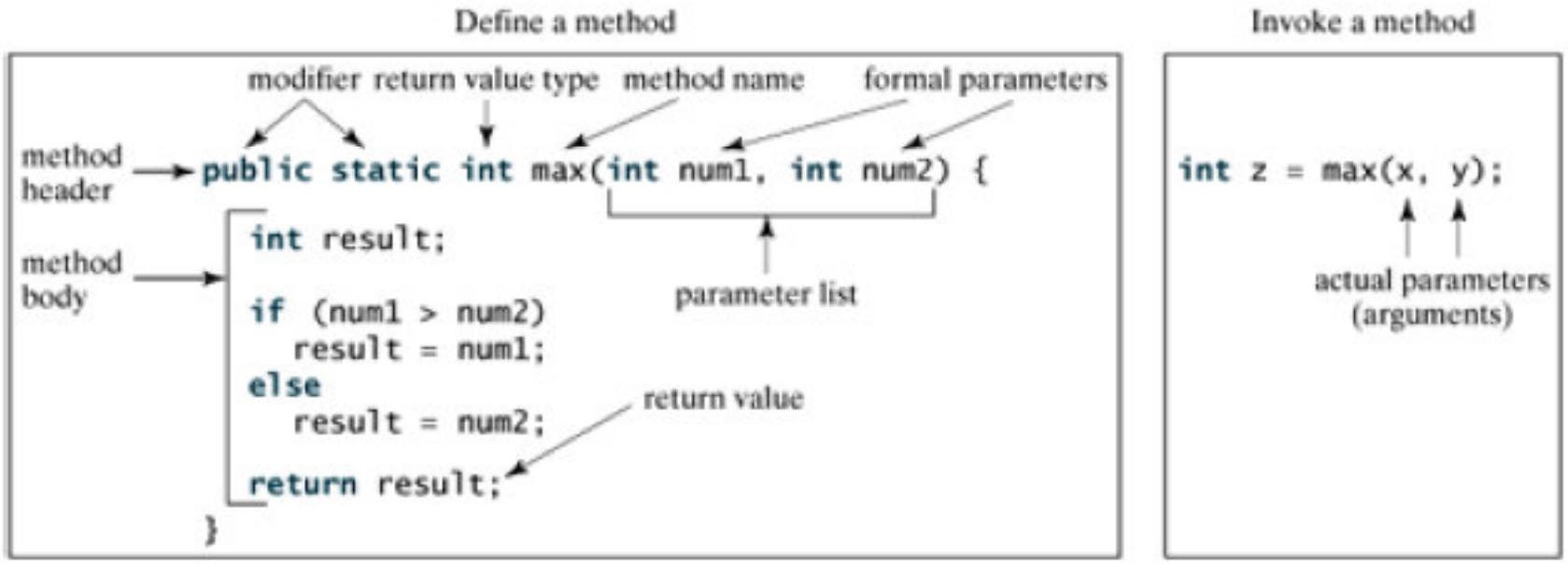
- Methods Called functions or procedures in other languages

- In general, a method has the following syntax:

```
modifier returnValueType methodName(list of parameters) {
        // Method body;
}
```

Creating a Method

- A method created to find which of two integers is bigger.



The Components of a Method

- Method declarations have six components, in order:
  - Modifiers
  - The return type
  - The method name
  - The parameters
  - An exception list
  - The method body

The Modifiers

- The ***modifier***, which is optional, tells the compiler how to call the method.

The Return Type

- A method may return a value.
- The returnValueType is the data type of the value the method returns.
- If the method does not return a value, the returnValueType is the keyword void.
- For example, the returnValueType in the main method is void.

The Return Type

- The method that returns a value is called a ***nonvoid method***, and the method that does not return a value is called a ***void method***.

- In other languages, a method with a nonvoid return value type is called a function, and a method with a void return value type is called a procedure.

The Method Name

- A method name can be any legal identifier
- By convention, the first (or only) word in a method name should be a verb in lowercase.
- Or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc.
- In multiword names, the first letter of each of the second and following words should be capitalized.

The Method Name

- Here are some examples:

  run

  runFast

  getBackground

  getFinalData

  compareTo

  setX

  isEmpty

The Parameters

- The variables defined in the method header are known as ***formal parameters***.

- When a method is invoked, you pass a value to the parameter. This value is referred to as ***actual parameter*** *or* ***argument***.

The Parameters

- You need to declare a separate data type for each parameter.

- For instance, int num1, num2 should be replaced by int num1, int num2.

Method signature

- Two of the components of a method declaration comprise the ***method signature***:
  - the method's name
  - the parameter list

- An example of a method declaration:

```
public double calculateAnswer(double wingSpan, int
    numberOfEngines, double length, double grossTons) {
    // do the calculation here
}
```

- The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

Method Body

- The ***method body*** contains a collection of statements that define what the method does.
- The method terminates when a return statement is executed.
- The keyword return is required for a nonvoid method to return a result.

Calling a Method

Calling a Method

- To use a method, you have to ***call*** or ***invoke*** it.
- There are two ways to call a method.
- If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(3, 4);
System.out.println(max(3, 4));
```

- If the method returns void, a call to the method must be a statement. For example:

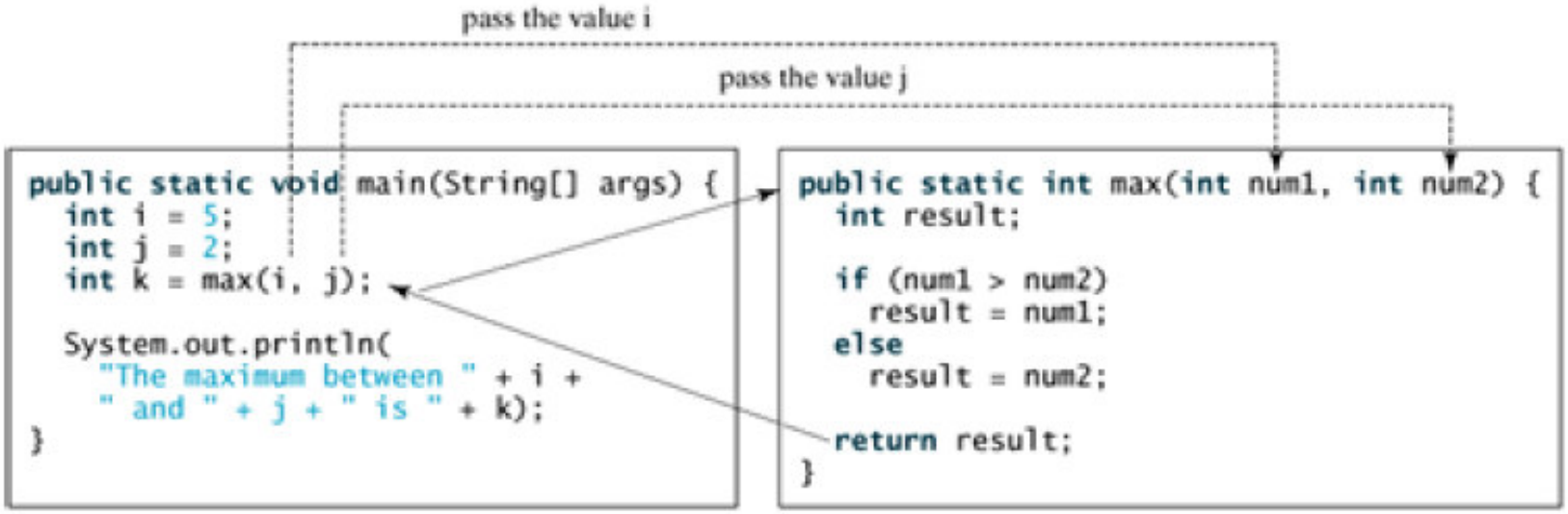
```
System.out.println("Welcome to Java!");
```

Calling a Method

```
1  public class TestMax {
2    /** Main method */
3    public static void main(String[] args) {
4      int i = 5;
5      int j = 2;
6      int k = max(i, j);
7      System.out.println("The maximum between " + i +
8          " and " + j + " is " + k);
9    }
10
11   /** Return the max between two numbers */
12   public static int max(int num1, int num2) {
13     int result;
14
15     if (num1 > num2)
16       result = num1;
17     else
18       result = num2;
19
20     return result;
21   }
22 }
```

Calling a Method

- When the **max** method is invoked, the flow of control transfers to the **max** method. Once the **max** method is finished, it returns the control back to the caller.



Caution

- A return statement is required for a nonvoid method.
- The method shown left below in (a) is logically correct, but it has a compilation error:

```
public static int sign(int n) {
  if (n > 0) return 1;
  else if (n == 0) return 0;
  else if (n < 0) return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0) return 1;
  else if (n == 0) return 0;
  else return -1;
}
```

(b)

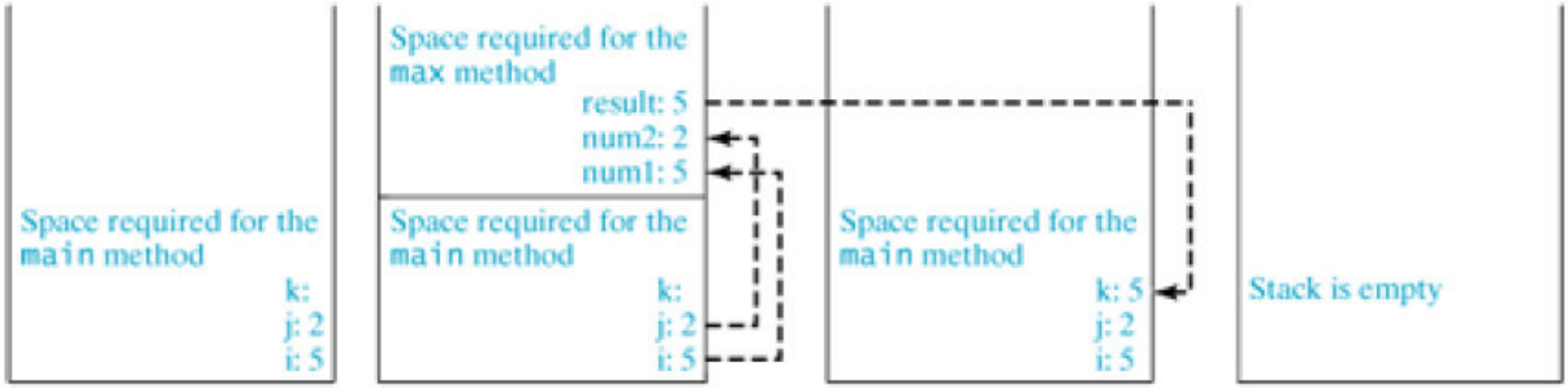
Reuse Methods from Other Classes

- One of the benefits of methods is for reuse.
- The max method can be invoked from any class besides TestMax.
- You can invoke the max method from other classes using ClassName.methodName (i.e., TestMax.max).

Call Stacks

- Each time a method is invoked, the system stores parameters and variables in an area of memory, known as a ***stack***, which stores elements in last-in first-out fashion.
- When a method calls another method, the caller's stack space is kept intact, and new space is created to handle the new method call.
- When a method finishes its work and returns to its caller, its associated space is released.

Call Stacks



(a) The `main` method is invoked.

(b) The `max` method is invoked.

(c) The `max` method is finished and the return value is sent to k.

(d) The `main` method is finished.

A void Method Example

```
1  public class TestVoidMethod {
2    public static void main(String[] args) {
3      printGrade(78.5);
4    }
5
6    public static void printGrade(double score) {
7      if (score < 0 || score > 100) {
8        System.out.println("Invalid score");
9        return;
10     }
11
12     if (score >= 90.0) {
13       System.out.println('A');
14     } else if (score >= 80.0) {
15       System.out.println('B');
16     } else if (score >= 70.0) {
17       System.out.println('C');
18     } else if (score >= 60.0) {
19       System.out.println('D');
20     } else {
21       System.out.println('F');
22     }
23   }
24 }
```

Passing Parameters

Passing Parameters

- The arguments must match the parameters in order, number, and compatible type, as defined in the method signature.
- When you invoke a method with a parameter, the value of the argument is passed to the parameter.
- This is referred to as ***pass-by-value***.

Passing Parameters

- If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter.
- The variable is not affected, regardless of the changes made to the parameter inside the method.

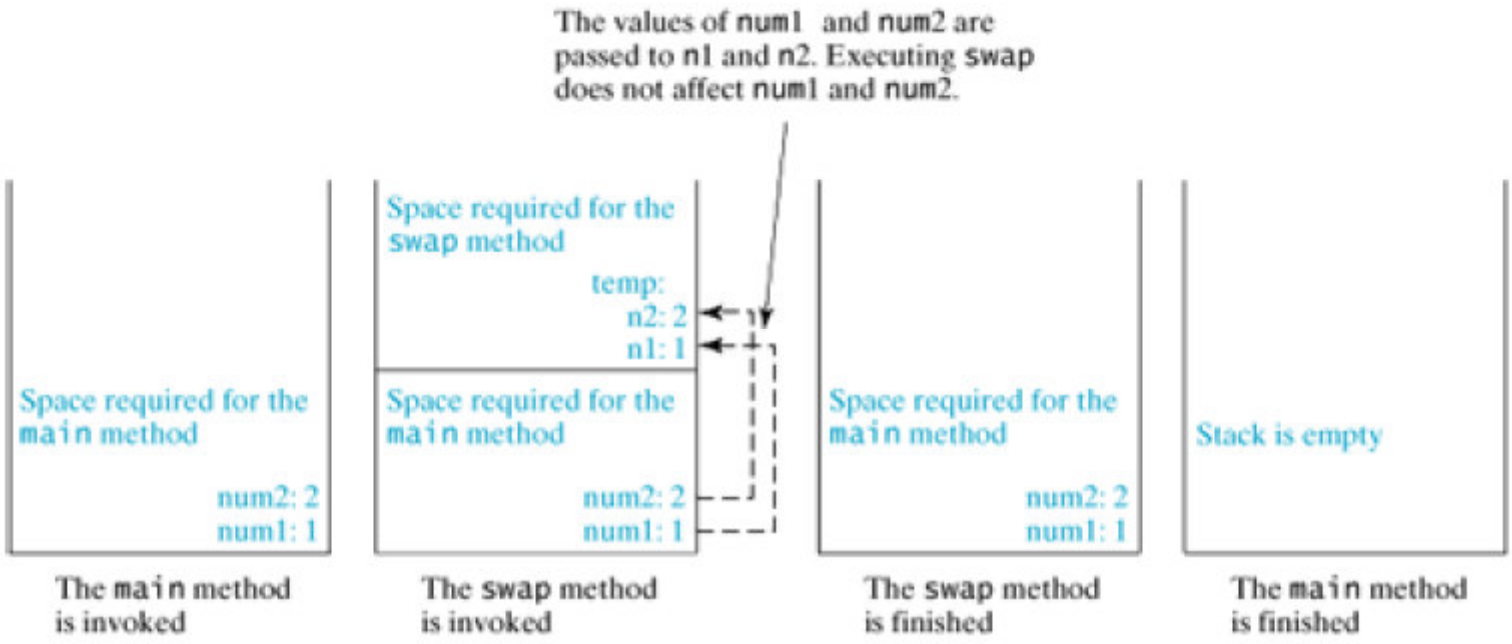
Passing Parameters

```
1  public class TestPassByValue {
2    /** Main method */
3    public static void main(String[] args) {
4      // Declare and initialize variables
5      int num1 = 1;
6      int num2 = 2;
7
8      System.out.println("Before invoking the swap method, num1 is " +
9          num1 + " and num2 is " + num2);
10
11     // Invoke the swap method to attempt to swap two variables
12     swap(num1, num2);
13
14     System.out.println("After invoking the swap method, num1 is " +
15         num1 + " and num2 is " + num2);
16   }
17
```

Passing Parameters

```
18      /** Swap two variables */
19      public static void swap(int n1, int n2) {
20          System.out.println("\tInside the swap method");
21          System.out.println("\t\tBefore swapping n1 is " + n1
22                  + " n2 is " + n2);
23
24          // Swap n1 with n2
25          int temp = n1;
26          n1 = n2;
27          n2 = temp;
28
29          System.out.println("\t\tAfter swapping n1 is " + n1
30                  + " n2 is " + n2);
31      }
32  }
```

Passing Parameters



Passing Parameters

- Another twist is to change the parameter name n1 in swap to num1.
- What effect does this have?
- No change occurs because it makes no difference whether the parameter and the argument have the same name.
- For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means passing the value of x to y.

Overloading Methods

Overloading Methods

- ***Method overloading*** is referred when two methods have the same name but different parameter lists within one class.
- Java can distinguish between methods with different method signatures.
- The Java compiler determines which method is used based on the method signature.

Overloading Methods

- The max method that was used earlier works only with the int data type.
- But what if you need to find which of two floating-point numbers has the maximum value?
- The solution is to create another method with the same name but different parameters.

```
public static double max(double num1, double num2)  {
        if (num1 > num2)
                return num1;
        else
                return num2;
}
```

Overloading Methods

```
1  public class TestMethodOverloading {
2    /** Main method */
3    public static void main(String[] args) {
4      // Invoke the max method with int parameters
5      System.out.println("The maximum between 3 and 4 is "
6          + max(3, 4));
7
8      // Invoke the max method with the double parameters
9      System.out.println("The maximum between 3.0 and 5.4 is "
10         + max(3.0, 5.4));
11
12     // Invoke the max method with three double parameters
13     System.out.println("The maximum between 3.0, 5.4, and 10.14 is "
14         + max(3.0, 5.4, 10.14));
15   }
16
17   /** Return the max between two int values */
18   public static int max(int num1, int num2) {
19     if (num1 > num2)
20       return num1;
21     else
22       return num2;
23   }
```

Overloading Methods

```
24
25    /** Find the max between two double values */
26    public static double max(double num1, double num2) {
27      if (num1 > num2)
28        return num1;
29      else
30        return num2;
31    }
32
33    /** Return the max among three double values */
34    public static double max(double num1, double num2, double num3) {
35      return max(max(num1, num2), num3);
36    }
37  }
```

Overloading Methods

- Can you invoke the max method with an int value and a double value, such as max(2, 2.5)?
- Yes, the max method for finding the maximum of two double values is invoked.
- The argument value 2 is automatically converted into a double value and passed to this method.

Ambiguous invocation

```
1  public class AmbiguousOverloading {
2    public static void main(String[] args) {
3      System.out.println(max(1, 2));
4    }
5
6    public static double max(int num1, double num2) {
7      if (num1 > num2)
8        return num1;
9      else
10       return num2;
11   }
12
13   public static double max(double num1, int num2) {
14     if (num1 > num2)
15       return num1;
16     else
17       return num2;
18   }
```

Ambiguous invocation

- Sometimes there are two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match.
- This is referred to as ***ambiguous invocation***.
- Ambiguous invocation causes a compilation error.

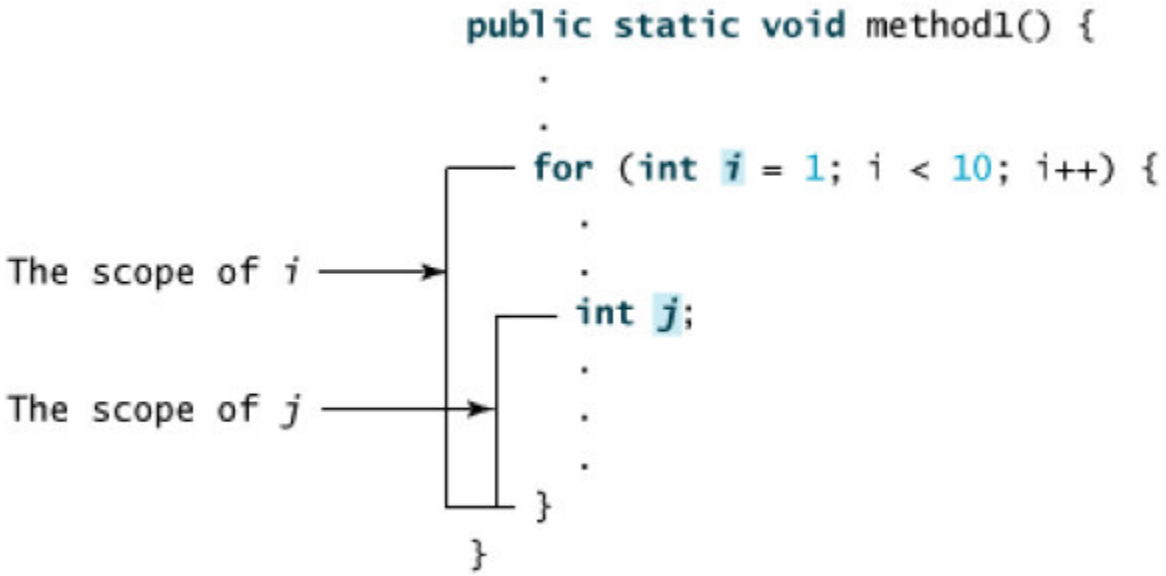
The Scope of Local Variables

The Scope of Local Variables

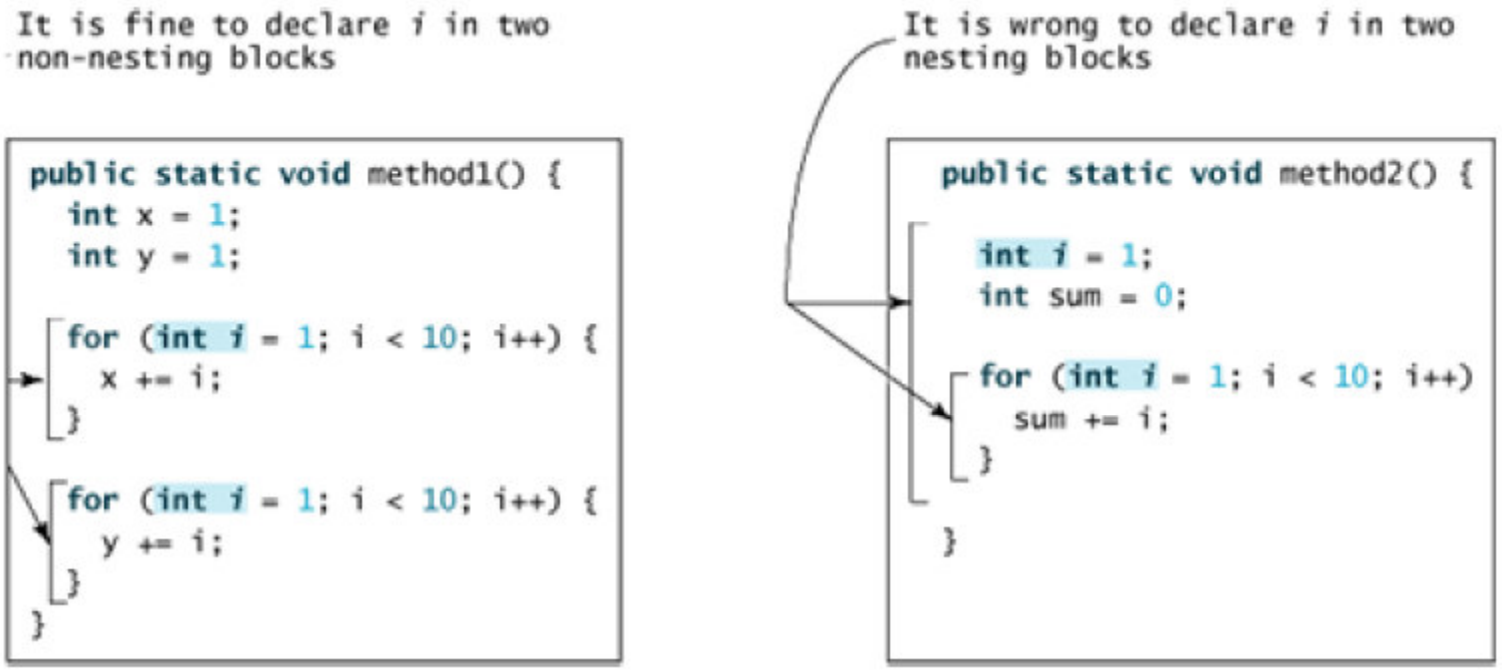
- The ***scope of a variable*** is the part of the program where the variable can be referenced.
- variable defined inside a method is referred to as a ***local variable***.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
- A parameter is actually a local variable. The scope of a method parameter covers the entire method.

The Scope of Local Variables

- A variable declared in the initial action part of a for loop header has its scope in the entire loop.
- But a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable



The Scope of Local Variables



- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks

The Scope of Variables

- Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

```
for (int i = 0; i < 10; i++) {
 }
System.out.println(i);
```

- The last statement would cause a syntax error because variable i is not defined outside of the for loop.

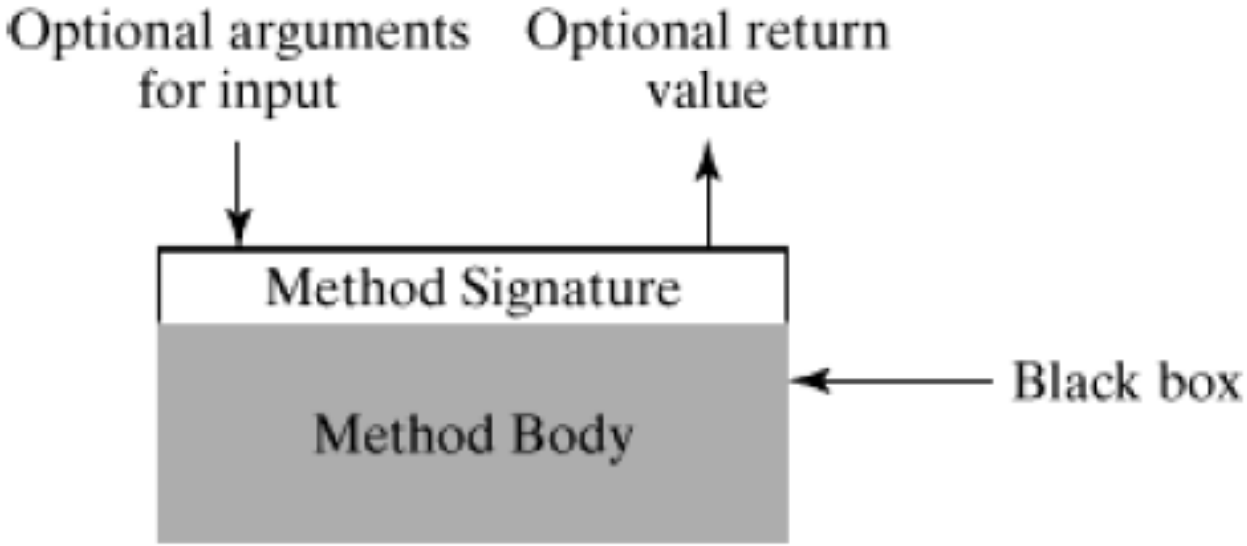
Method Abstraction

Method Abstraction

- ***Method abstraction*** is achieved by separating the use of a method from its implementation.
- The details of the implementation are encapsulated in the method and hidden from the client who invokes the method.
- This is known as ***information hiding*** or ***encapsulation***.
- If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature.

Method Abstraction

- You can think of the method body as a black box that contains the detailed implementation for the method.



Method Abstraction

- You have already used the System.out.println method to print.
- You know how to write the code to invoke this method in your program, but as a user of this method, you are not required to know how it is implemented.

Method Abstraction

- The concept of method abstraction can be applied to the process of developing programs.
- When writing a large program, you can use the "***divide and conquer***" strategy, also known as ***stepwise refinement***, to decompose it into subproblems.
- The subproblems can be further decomposed into smaller, more manageable problems.

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 5)
- S. Zakhour and et el., **The Java Tutorial: A Short Course on the Basics**, 4th Edition, Prentice Hall, 2006. (Chapter 4)

The End