

Chapter 6 Methods

Objectives

- To define methods with formal parameters (§6.2).
- To invoke methods with actual parameters (i.e., arguments) (§6.2).
- To define methods with a return value (§6.3).
- To define methods without a return value (§6.4).
- To pass arguments by value (§6.5).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.6).
- To write a method that converts hexadecimal to decimals (§6.7).
- To use method overloading and understand ambiguous overloading (§6.8).
- To determine the scope of variables (§6.9).
- To apply the concept of method abstraction in software development (§6.10).
- To design and implement methods using stepwise refinement (§6.10).

Defining Methods

```
public class TestMax {  
    /** Main method */  
    public static void main(String[] args) {  
        int i = 5;  
        int j = 2;  
        int k = max(i, j);  
        System.out.println("The maximum of " + i +  
            " and " + j + " is " + k);  
    }  
  
    /** Return the max of two numbers */  
    public static int max(int num1, int num2) {  
        int result;  
  
        if (num1 > num2)  
            result = num1;  
        else  
            result = num2;  
  
        return result;  
    }  
}
```

Defining Methods

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

Define a method

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

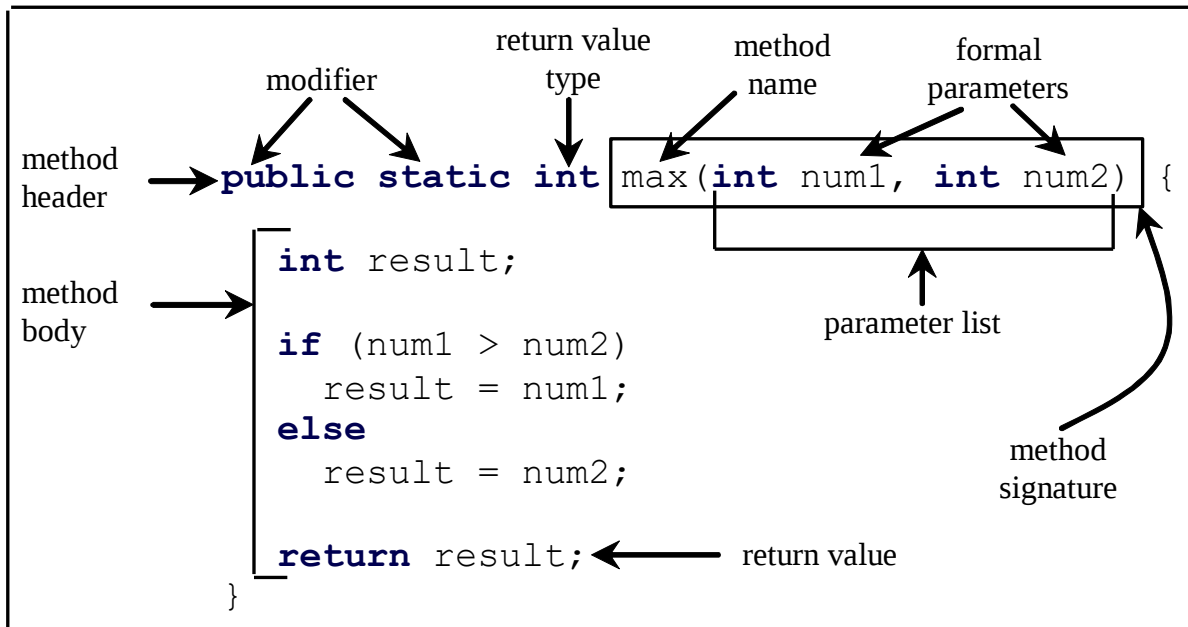
Invoke a method

```
int z = max(x, y);  
         ↑  ↑  
    actual parameters  
    (arguments)
```

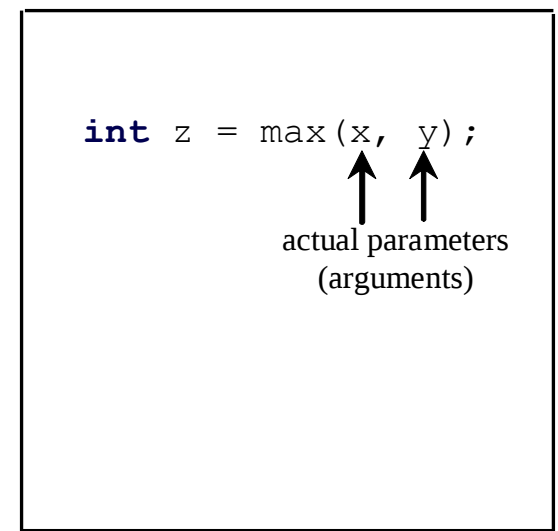
Defining Methods

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

Define a method



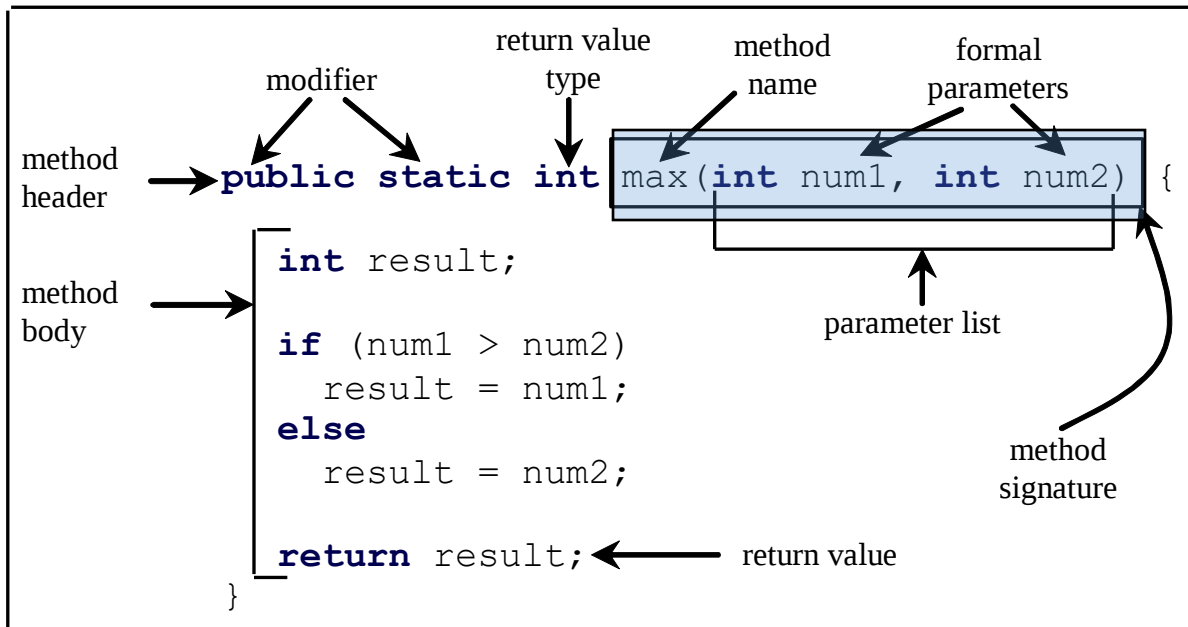
Invoke a method



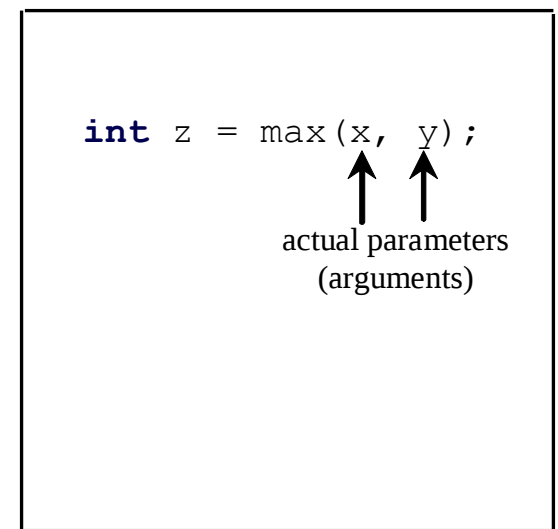
Method Signature

Method signature is the combination of the method name and the parameter list.

Define a method



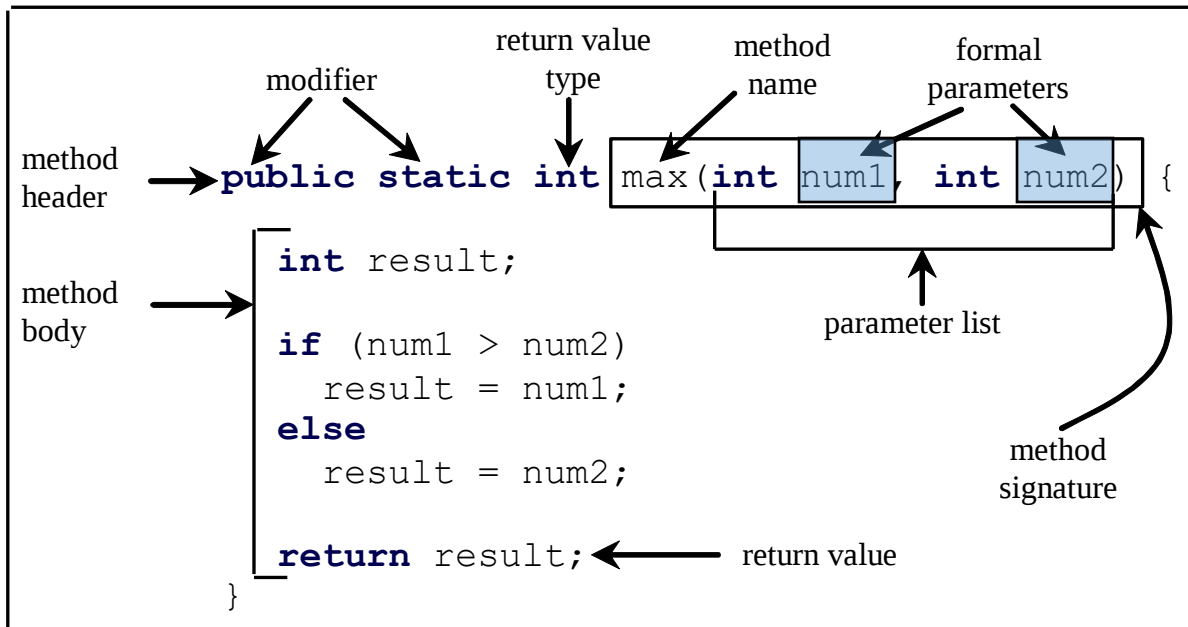
Invoke a method



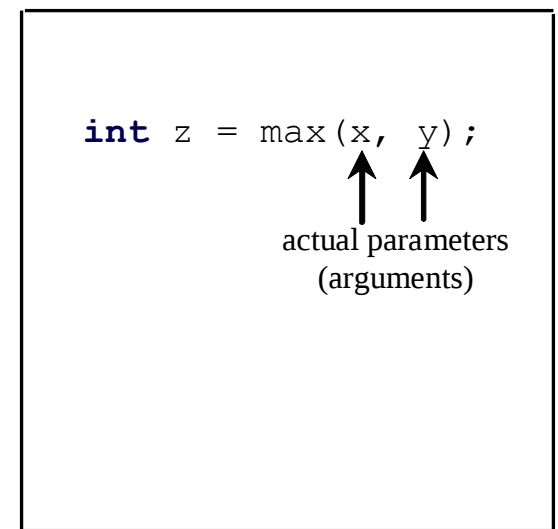
Formal Parameters

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

Define a method



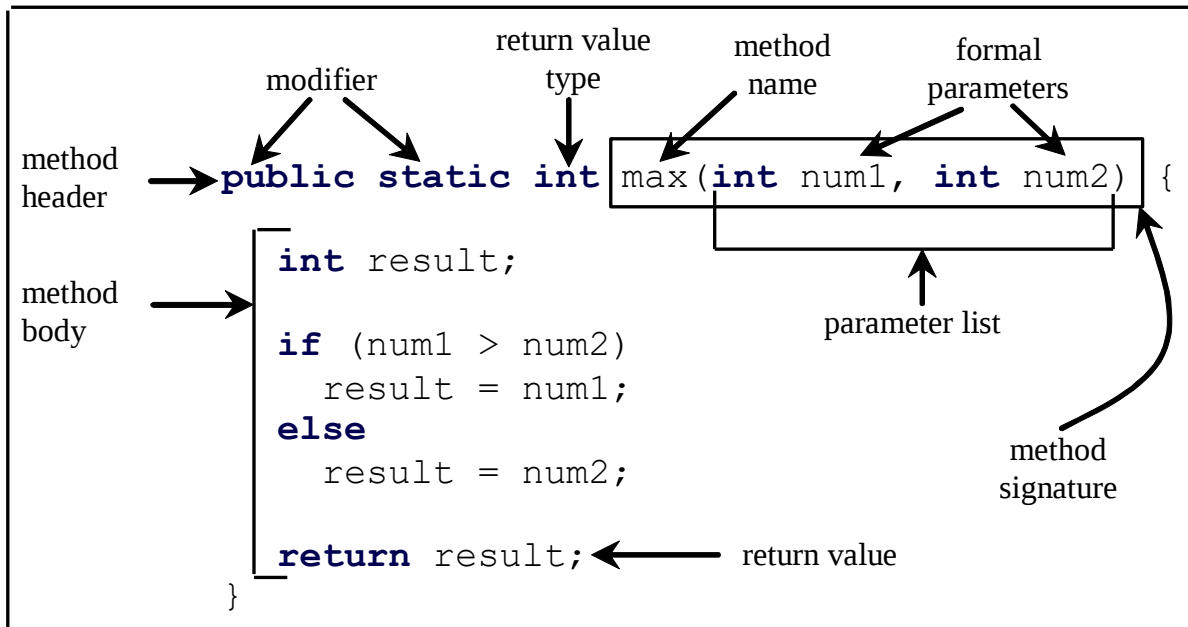
Invoke a method



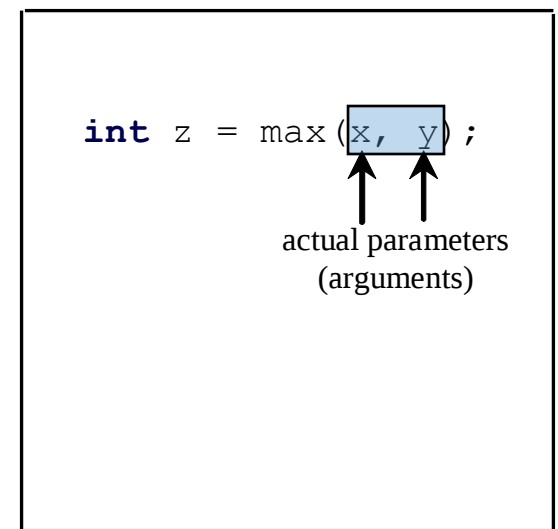
Actual Parameters

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

Define a method



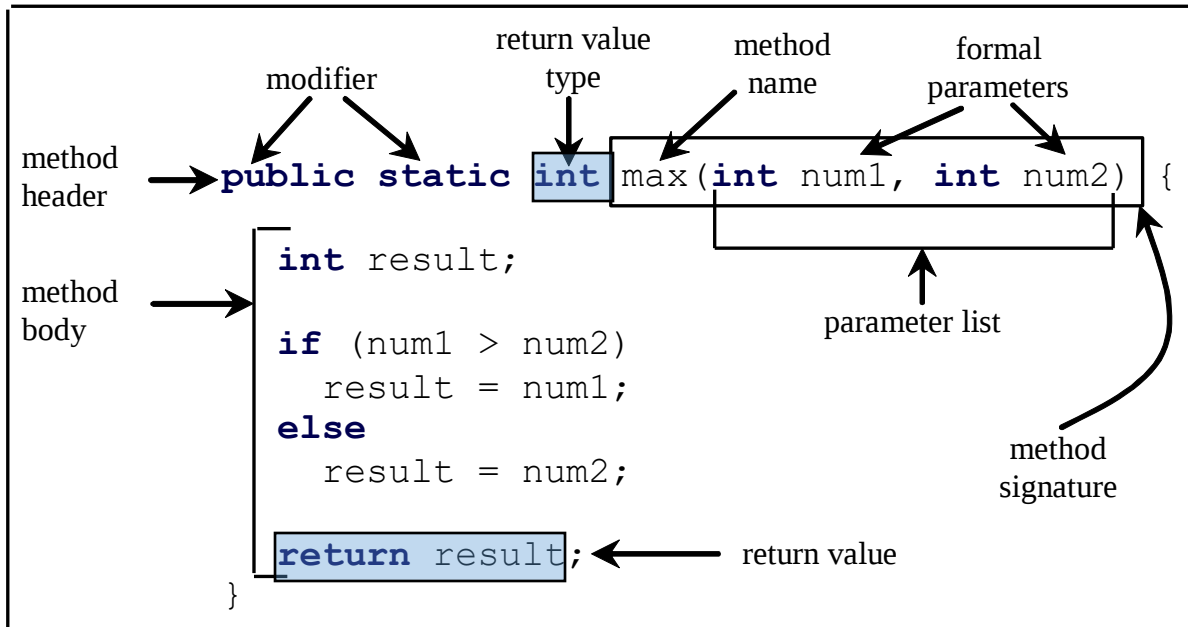
Invoke a method



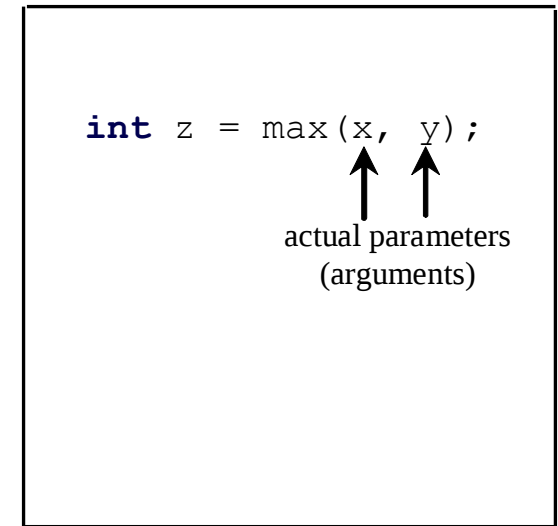
Return Value Type

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.

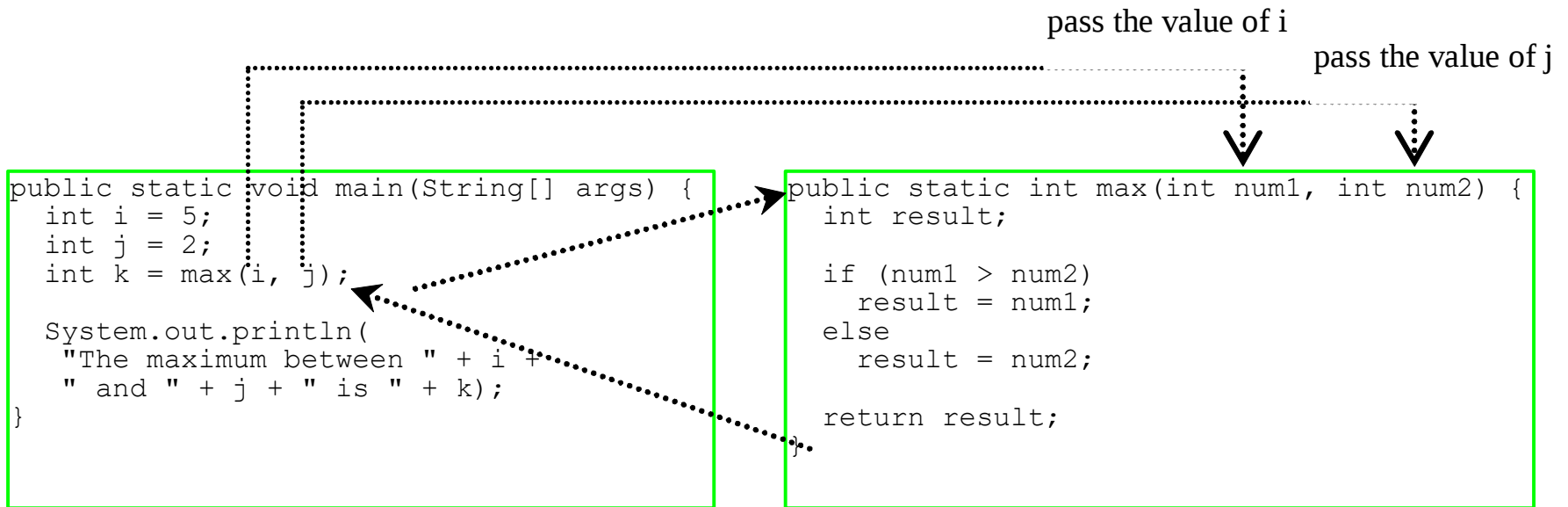
Define a method



Invoke a method



Calling Methods



CAUTION

A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
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)

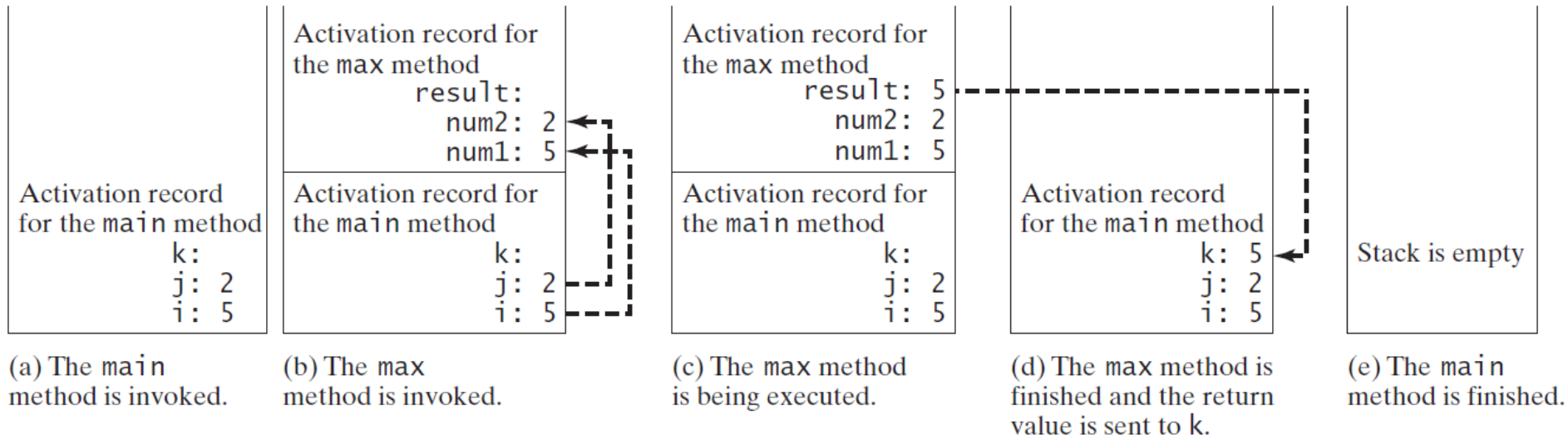
To fix this problem, delete if ($n < 0$) in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

Reuse Methods from Other Classes

NOTE: One of the benefits of methods is for reuse. The max method can be invoked from any class besides TestMax. If you create a new class Test, you can invoke the **static method** max using ClassName.methodName (e.g., TestMax.max).

Call Stacks

Each time a method is invoked, the system creates an activation record (also called an activation frame) that **stores parameters and variables for the method** and places the activation record in an area of memory known as a call stack. A call stack is also known as an execution stack, runtime stack, or machine stack, and it is often shortened to just “the stack.” When a method calls another method, the caller’s activation record is kept intact, and a new activation record is created for the new method called. When a method finishes its work and returns to its caller, its activation record is removed from the call stack.



Trace Call Stack

i is declared and initialized

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

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

i: 5

method

Trace Call Stack

j is declared and initialized

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

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

j: 2
i: 5

method

Trace Call Stack

Declare k

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

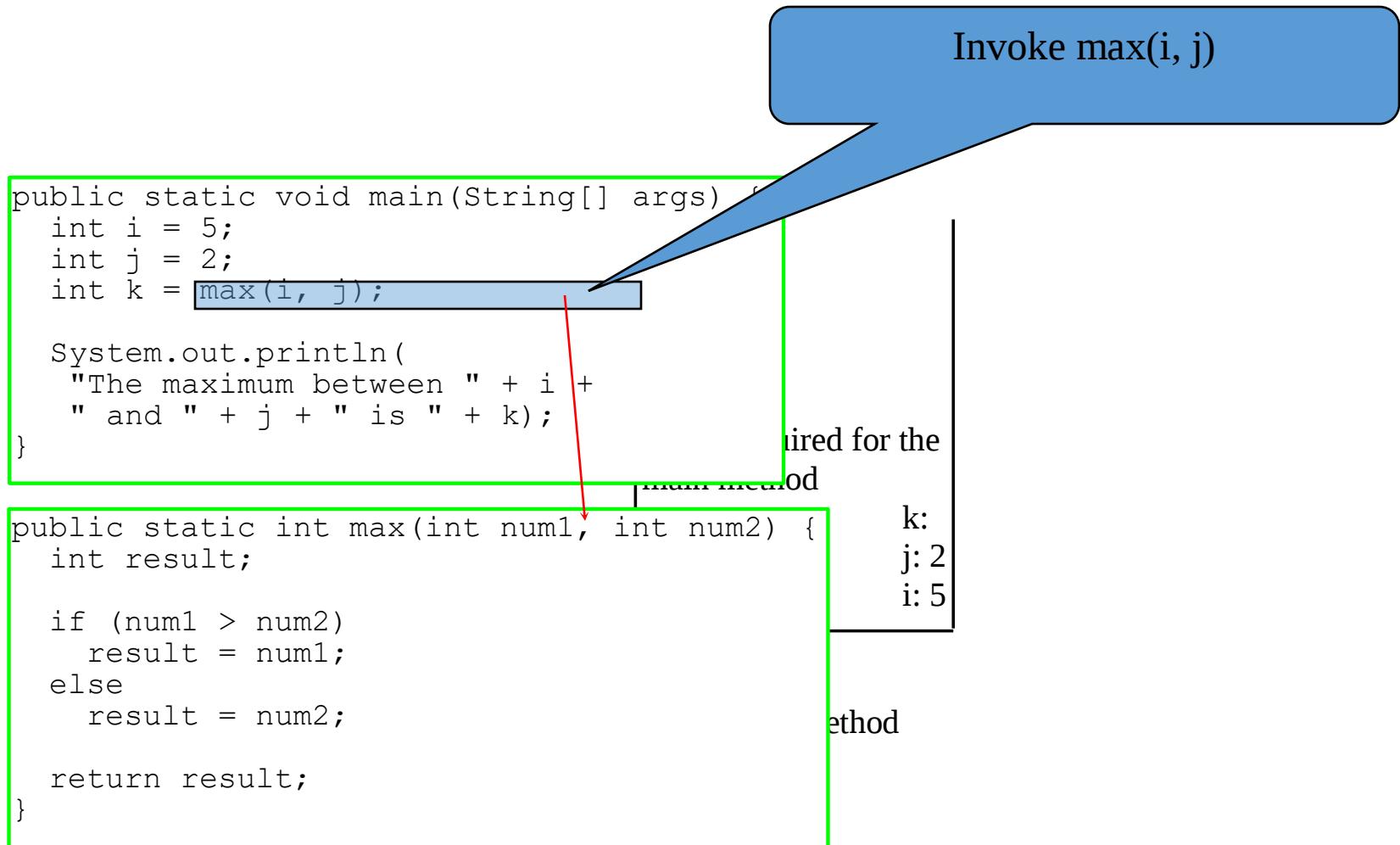
```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

required for the
method

k:
j: 2
i: 5

method

Trace Call Stack



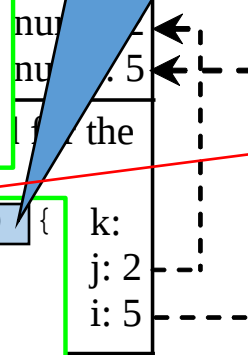
Trace Call Stack

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

main method

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

pass the values of i and j to num1
and num2



is

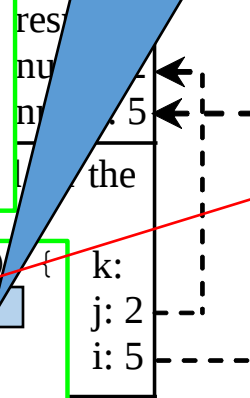
Trace Call Stack

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

main method

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

Declare result



is

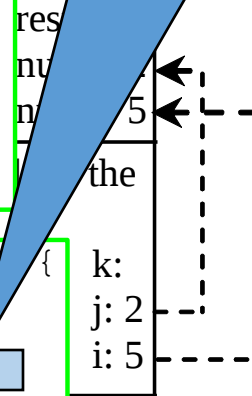
Trace Call Stack

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

main method

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

(num1 > num2) is true



is

Trace Call Stack

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

main method

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

Assign num1 to result

l for
resu
nu
nu
l f
le
k:
j: 2
i: 5

l is

Trace Call Stack

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

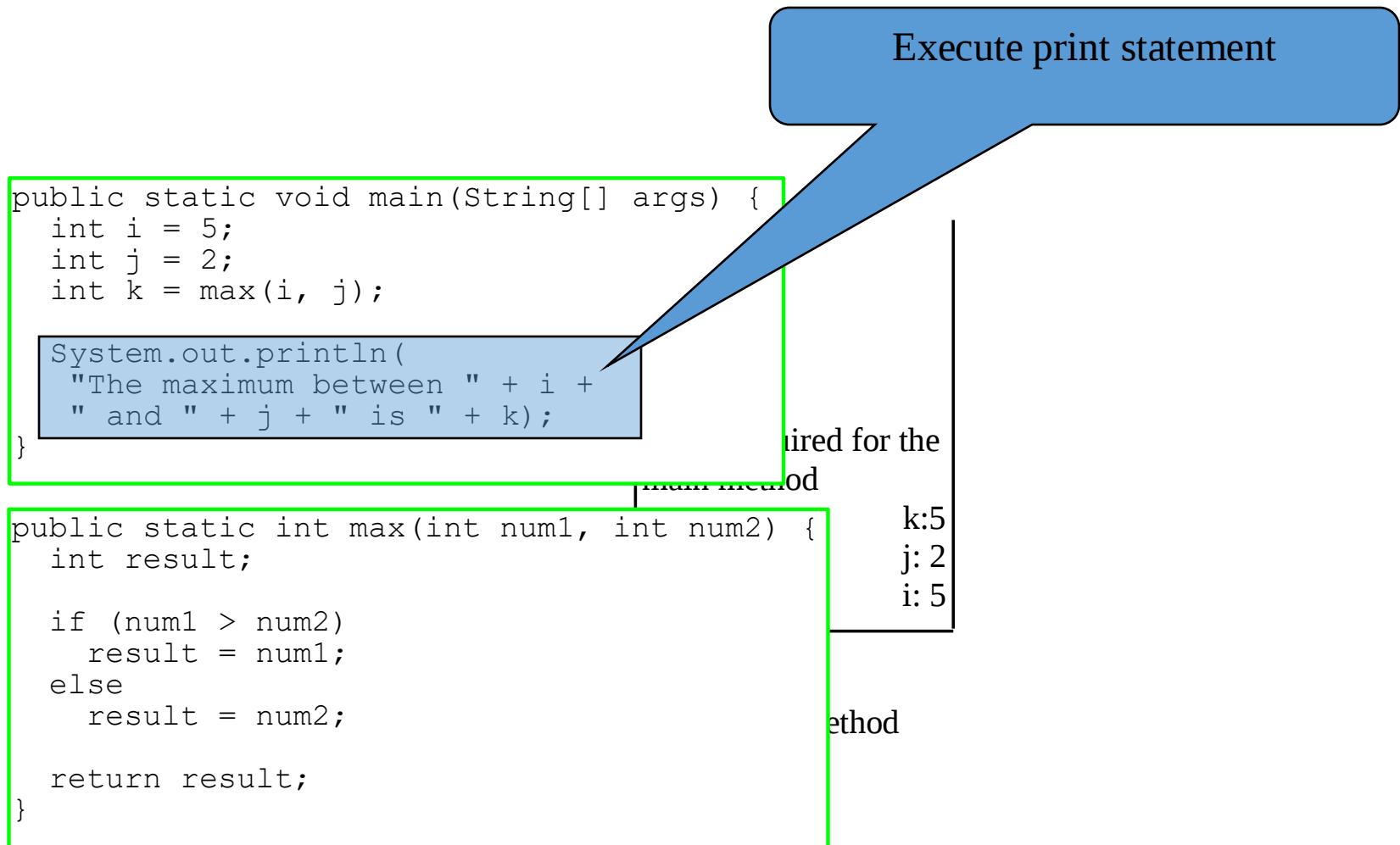
main method

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

Return result and assign it to k

k:5
j: 2
i: 5

Trace Call Stack



void Method Example

```
public class TestVoidMethod {  
    public static void main(String[] args) {  
        System.out.print("The grade is ");  
        printGrade(78.5);  
  
        System.out.print("The grade is ");  
        printGrade(59.5);  
    }  
  
    public static void printGrade(double score) {  
        if (score >= 90.0) {  
            System.out.println('A');  
        }  
        else if (score >= 80.0) {  
            System.out.println('B');  
        }  
        else if (score >= 70.0) {  
            System.out.println('C');  
        }  
        else if (score >= 60.0) {  
            System.out.println('D');  
        }  
        else {  
            System.out.println('F');  
        }  
    }  
}
```

Passing Parameters

```
public static void nPrintln(String message, int n) {  
    for (int i = 0; i < n; i++)  
        System.out.println(message);  
}
```

You can use `nPrintln("Hello", 3)` to print `Hello` three times. The `nPrintln("Hello", 3)` statement passes the actual string parameter `Hello` to the parameter `message`, passes `3` to `n`, and prints `Hello` three times. However, the statement `nPrintln(3, "Hello")` would be wrong. The data type of `3` does not match the data type for the first parameter, `message`, nor does the second argument, `Hello`, match the second parameter, `n`.

Pass by Value

When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. 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

Pass by Value

```
1 public class Increment {  
2     public static void main(String[] args) {  
3         int x = 1;  
4         System.out.println("Before the call, x is " + x);  
5         increment(x);  
6         System.out.println("After the call, x is " + x);  
7     }  
8  
9     public static void increment(int n) {  
10        n++;  
11        System.out.println("n inside the method is " + n);  
12    }  
13 }
```

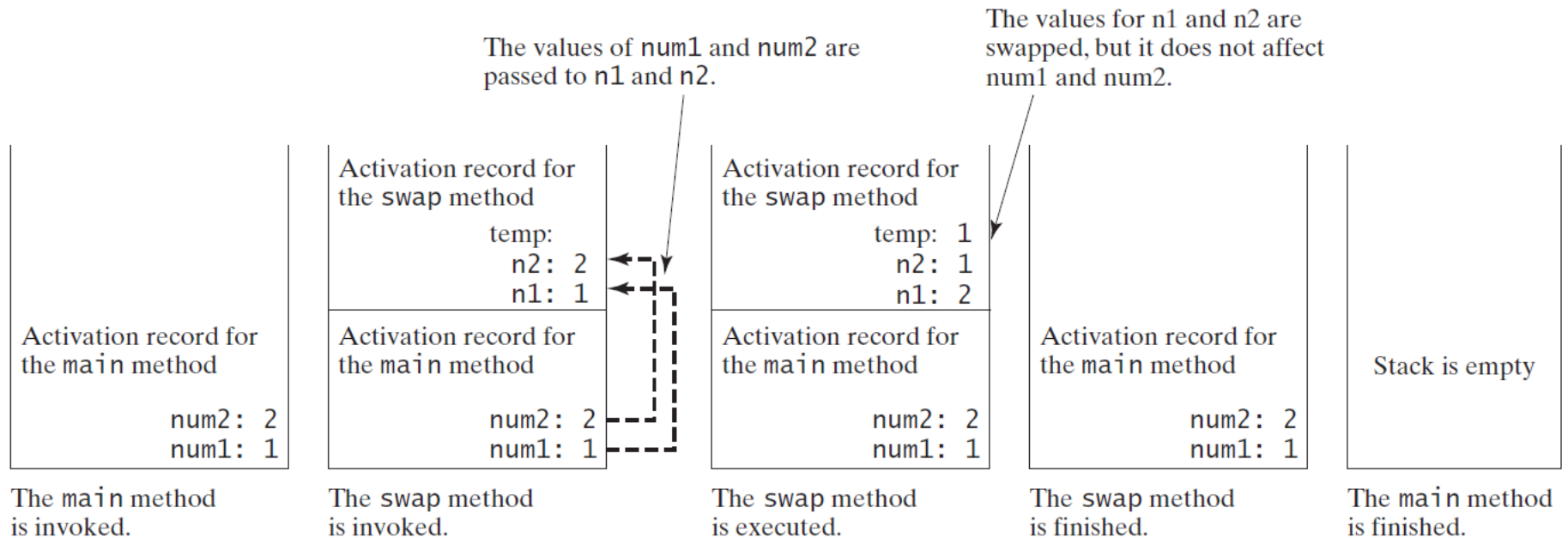
Before the call, x is 1
n inside the method is 2
After the call, x is 1

Pass by Value

```
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
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             + " and 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             + " and n2 is " + n2);
31     }
32 }
```

false swap

Pass by Value, cont.



Overloading Methods

- *Overloading methods enables you to define the methods with the same name as long as their signatures are different.*
- The **max** method that was used earlier works only with the **int** data type. But what if you need to determine which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

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

Ambiguous Invocation

- The Java compiler determines which method to use based on the method signature.
- Sometimes there may be 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 is a compile error.

Ambiguous Invocation

```
public class AmbiguousOverloading {  
    public static void main(String[] args) {  
        System.out.println(max(1, 2));  
    }  
  
    public static double max(int num1, double num2) {  
        if (num1 > num2)  
            return num1;  
        else  
            return num2;  
    }  
  
    public static double max(double num1, int num2) {  
        if (num1 > num2)  
            return num1;  
        else  
            return num2;  
    }  
}
```

Both `max(int, double)` and `max(double, int)` are possible candidates to match `max(1, 2)`. Because neither is better than the other, the invocation is ambiguous, resulting in a compile error.

Scope of Local Variables

A **local variable**: a variable defined inside a method.

Scope: the part of the program where the variable can be referenced.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

Scope of Local Variables, cont.

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.

Scope of Local Variables, cont.

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 and to the end of the block that contains the variable.

```
public static void method1() {  
    .  
    .  
    for (int i = 1; i < 10; i++) {  
        .  
        .  
        int j;  
        .  
        .  
        .  
    }  
}
```

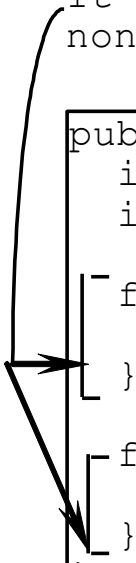
The scope of i →

The scope of j →

Scope of Local Variables, cont.

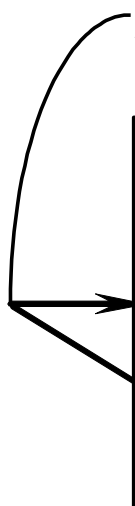
It is fine to declare `i` in two non-nesting blocks

```
public static void method1() {  
    int x = 1;  
    int y = 1;  
  
    for (int i = 1; i < 10; i++) {  
        x += i;  
    }  
  
    for (int i = 1; i < 10; i++) {  
        y += i;  
    }  
}
```



It is wrong to declare `i` in two nesting blocks

```
public static void method2() {  
    int i = 1;  
    int sum = 0;  
  
    for (int i = 1; i < 10; i++) {  
        sum += i;  
    }  
}
```



Scope of Local Variables, cont.

// Fine with no errors

```
public static void correctMethod() {  
    int x = 1;  
    int y = 1;  
    // i is declared  
    for (int i = 1; i < 10; i++) {  
        x += i;  
    }  
    // i is declared again  
    for (int i = 1; i < 10; i++) {  
        y += i;  
    }  
}
```

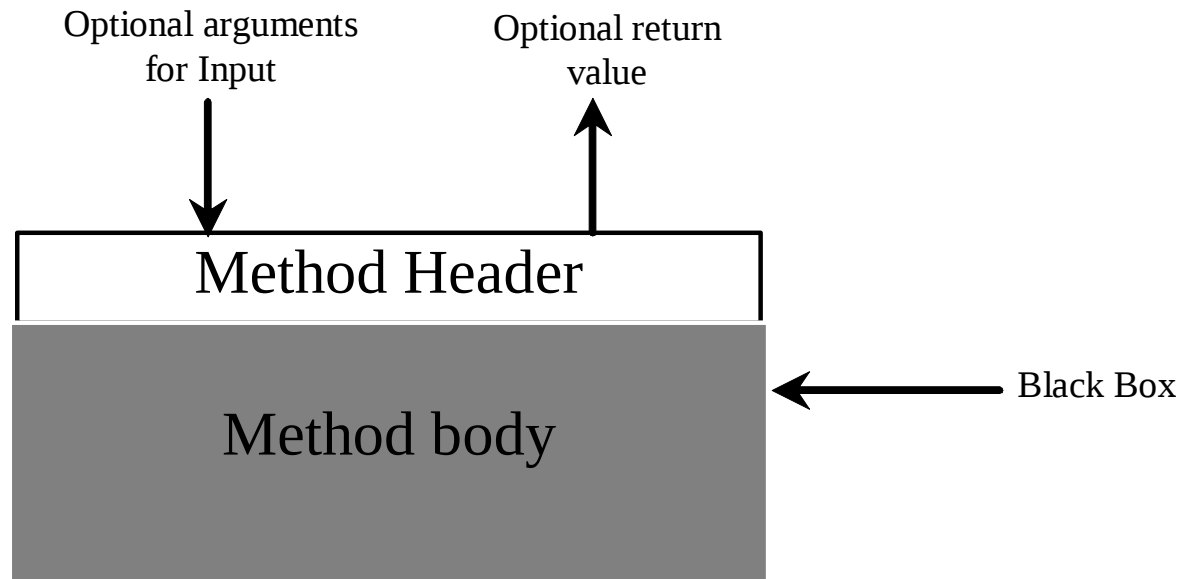
Scope of Local Variables, cont.

// With errors

```
public static void incorrectMethod() {  
    int x = 1;  
    int y = 1;  
    for (int i = 1; i < 10; i++) {  
        int x = 0;  
        x += i;  
    }  
}
```

Method Abstraction

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



Benefits of Methods

- Write a method once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

Assignment

By use of Methods, write Java programs to find out the area of;

1. Circle
2. Triangle

The program should prompt the user to enter for example the **Length**, **Height** and or **Radius** then goes ahead to display the area of the object respectively.

Case Study: Generating Random Characters, cont.

Now let us consider how to generate a random lowercase letter. The Unicode for lowercase letters are consecutive integers starting from the Unicode for 'a', then for 'b', 'c', ..., and 'z'. The Unicode for 'a' is

`(int)'a'`

So, a random integer between `(int)'a'` and `(int)'z'` is

`(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))`

Case Study: Generating Random Characters, cont.

As discussed in Chapter 2, all numeric operators can be applied to the char operands. The char operand is cast into a number if the other operand is a number or a character. So, the preceding expression can be simplified as follows:

```
'a' + Math.random() * ('z' - 'a' + 1)
```

So a random lowercase letter is

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

Case Study: Generating Random Characters, cont.

To generalize the foregoing discussion, a random character between any two characters `ch1` and `ch2` with `ch1 < ch2` can be generated as follows:

```
(char)(ch1 + Math.random() * (ch2 - ch1 + 1))
```

The RandomCharacter Class

```
// RandomCharacter.java: Generate random characters

public class RandomCharacter {

    /** Generate a random character between ch1 and ch2 */
    public static char getRandomCharacter(char ch1, char ch2) {
        return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
    }

    /** Generate a random lowercase letter */
    public static char getRandomLowerCaseLetter() {
        return getRandomCharacter('a', 'z');
    }

    /** Generate a random uppercase letter */
    public static char getRandomUpperCaseLetter() {
        return getRandomCharacter('A', 'Z');
    }

    /** Generate a random digit character */
    public static char getRandomDigitCharacter() {
        return getRandomCharacter('0', '9');
    }

    /** Generate a random character */
    public static char getRandomCharacter() {
        return getRandomCharacter('\u0000', '\uFFFF');
    }
}
```

The TestRandomCharacter Class

```
1 public class TestRandomCharacter {
2     /** Main method */
3     public static void main(String[] args) {
4         final int NUMBER_OF_CHARS = 175;
5         final int CHARS_PER_LINE = 25;
6
7         // Print random characters between 'a' and 'z', 25 chars per line
8         for (int i = 0; i < NUMBER_OF_CHARS; i++) {
9             char ch = RandomCharacter.getRandomLowerCaseLetter();
10            if ((i + 1) % CHARS_PER_LINE == 0)
11                System.out.println(ch);
12            else
13                System.out.print(ch);
14        }
15    }
16 }
```

```
gmjsohezfkgtazqgmswfc1rao
pnrunulnwmaztlfjedmpchcif
1alqdgivxkxpbzulrmqmbhikr
1bnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatdslvbwbz
bfesoklwbhnooygiigzdxuqni
```