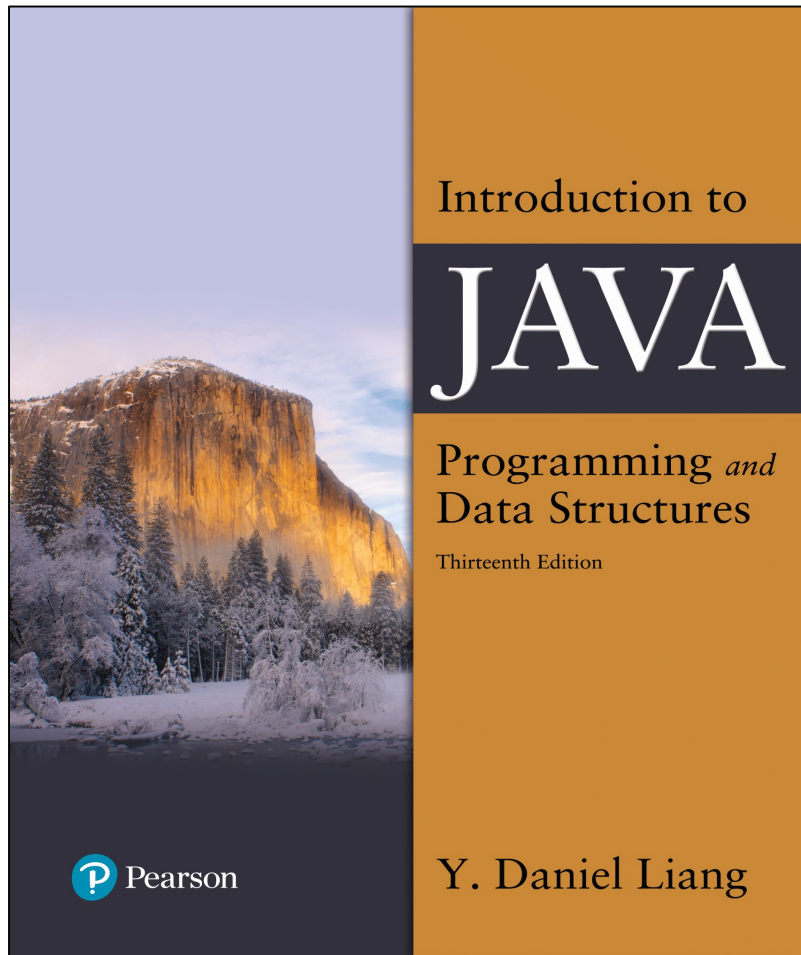


Introduction to Java Programming and Data Structures

Thirteenth Edition



Chapter 6

Methods

Opening Problem

Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

Problem (1 of 2)

```
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum += i;
System.out.println("Sum from 1 to 10 is " + sum);
```

```
sum = 0;
for (int i = 20; i <= 30; i++)
    sum += i;
System.out.println("Sum from 20 to 30 is " + sum);
```

```
sum = 0;
for (int i = 35; i <= 45; i++)
    sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Objectives (1 of 2)

- 6.1** To define methods with formal parameters (§6.2).
- 6.2** To invoke methods with actual parameters (i.e., arguments) (§6.2).
- 6.3** To define methods with a return value (§6.3).
- 6.4** To define methods without a return value (§6.4).
- 6.5** To pass arguments by value (§6.5).
- 6.6** To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.6).
- 6.7** To write a method that converts hexadecimal to decimals (§6.7).

Objectives (2 of 2)

6.8 To use method overloading and understand ambiguous overloading (§6.8).

6.9 To determine the scope of variables (§6.9).

6.10 To apply the concept of method abstraction in software development (§6.10).

6.11 To design and implement methods using stepwise refinement (§6.11).

6.1 To define methods with formal parameters (§6.2).

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

Define a method

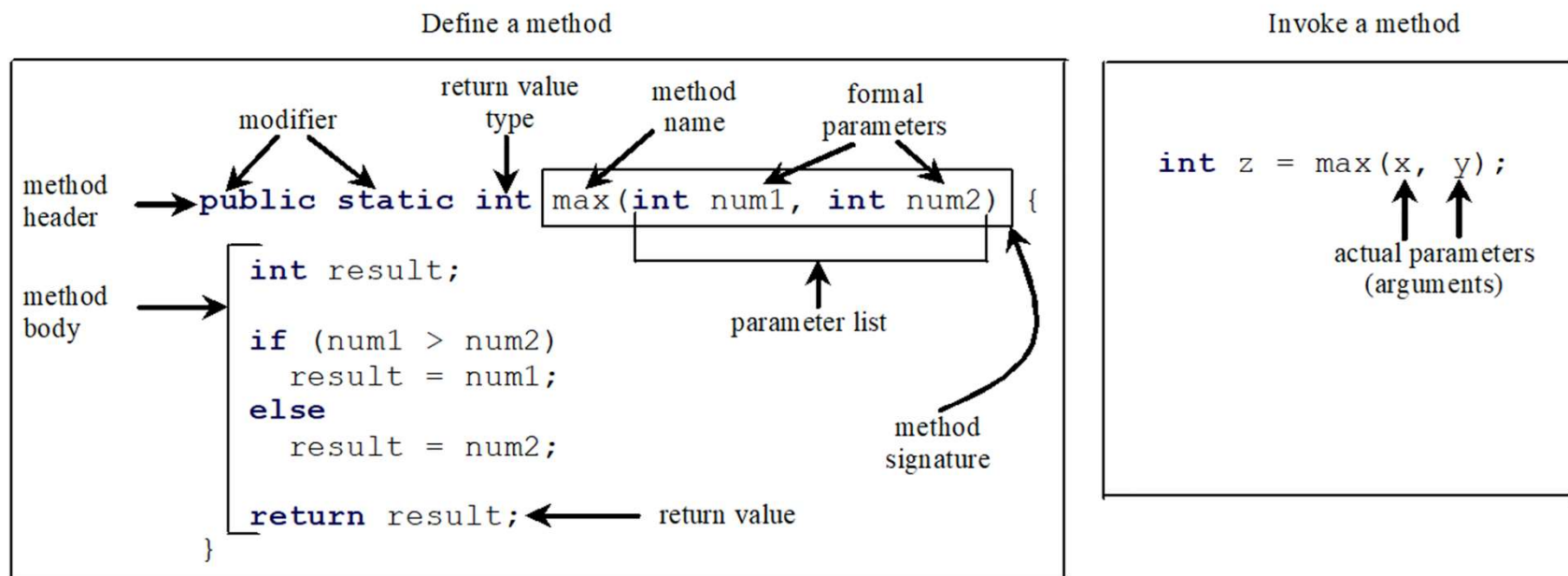
```
public static int max(int num1, int num2) {  
    int result;  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    return result;  
}
```

Invoke a method

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

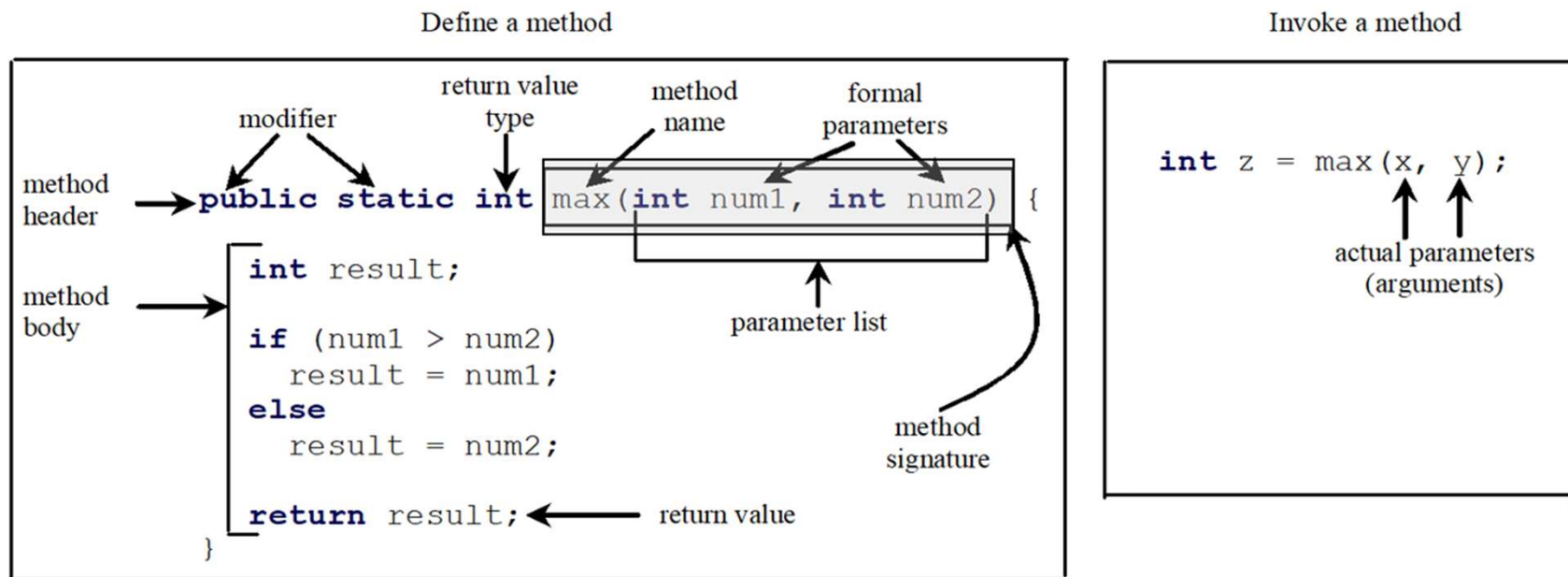
Defining Methods (2 of 2)

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



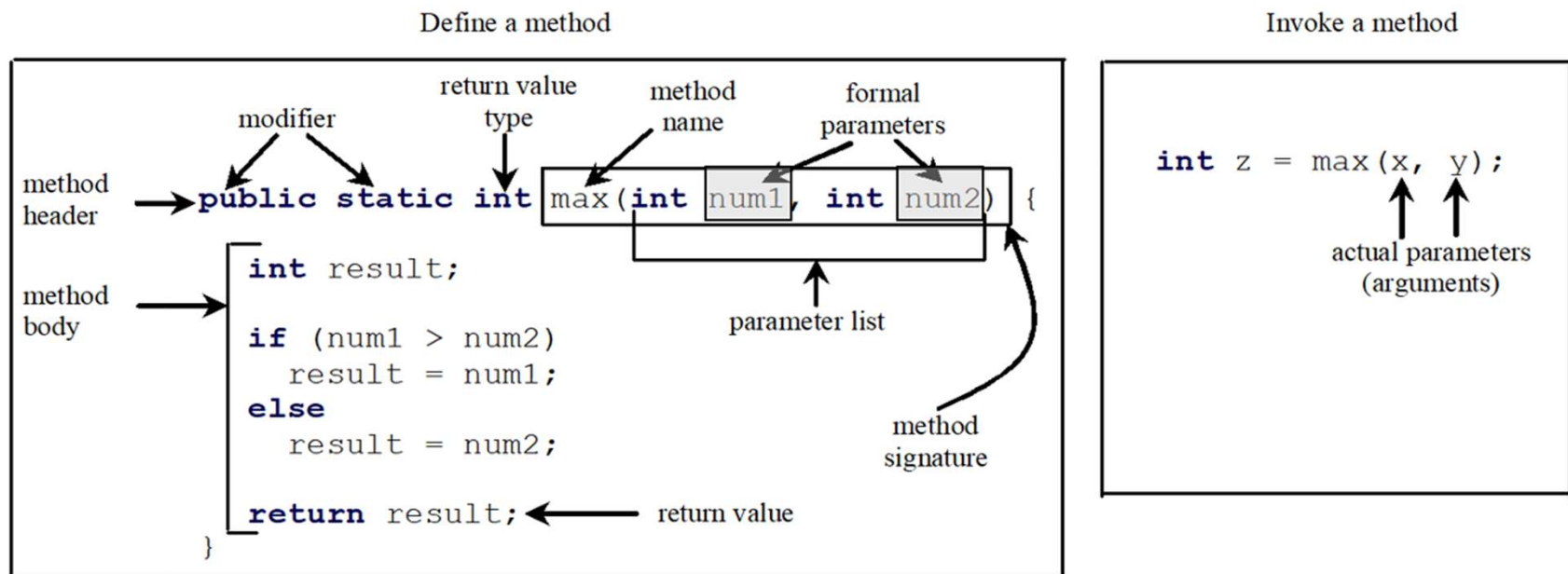
Method Signature

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



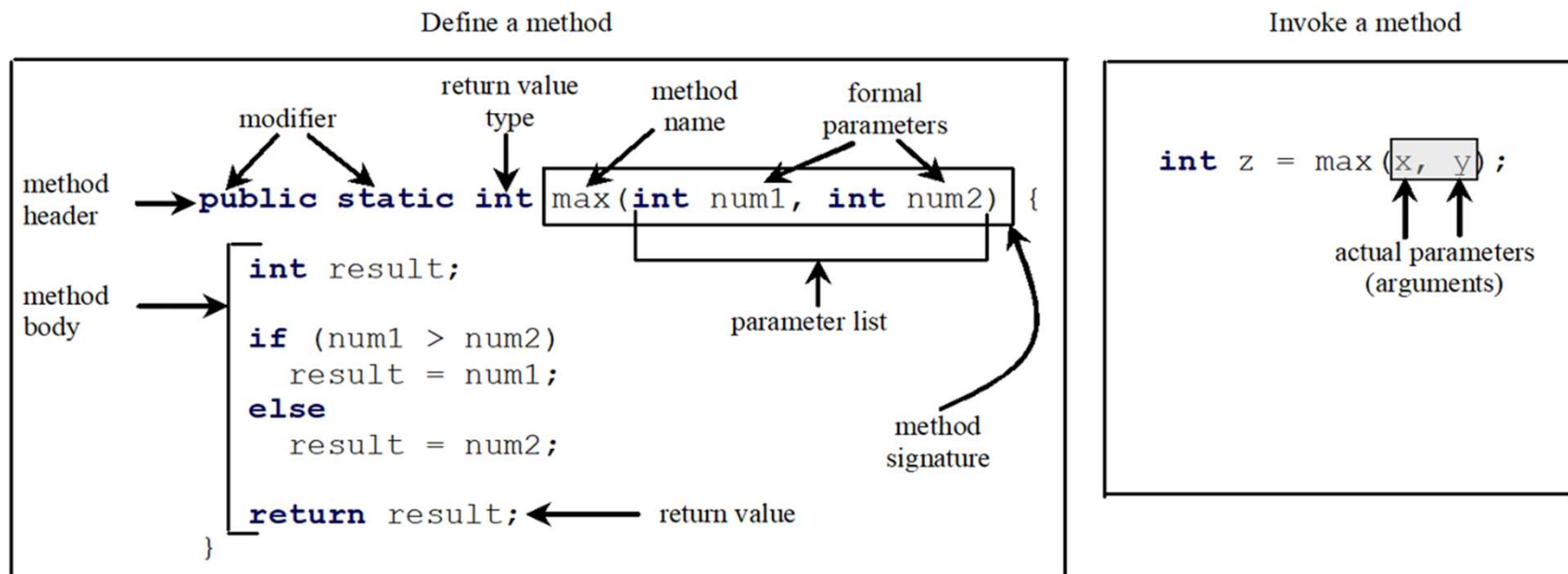
Formal Parameters

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



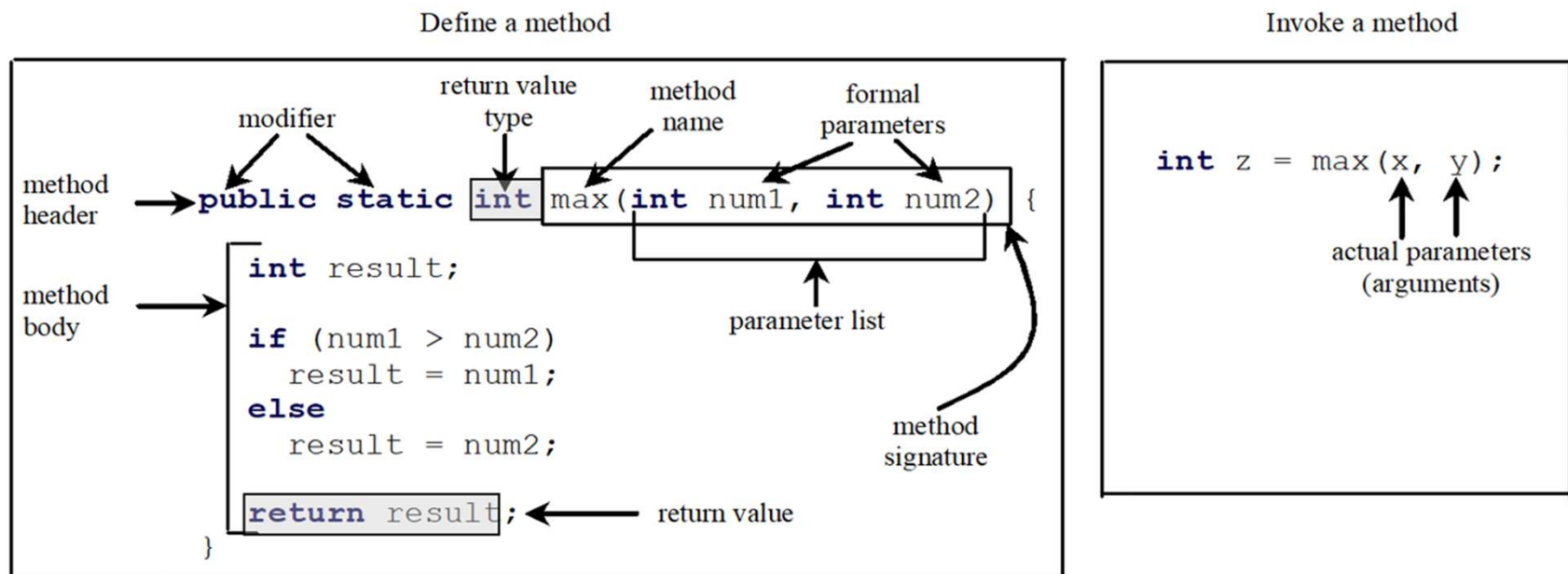
Actual Parameters

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



Return Value Type

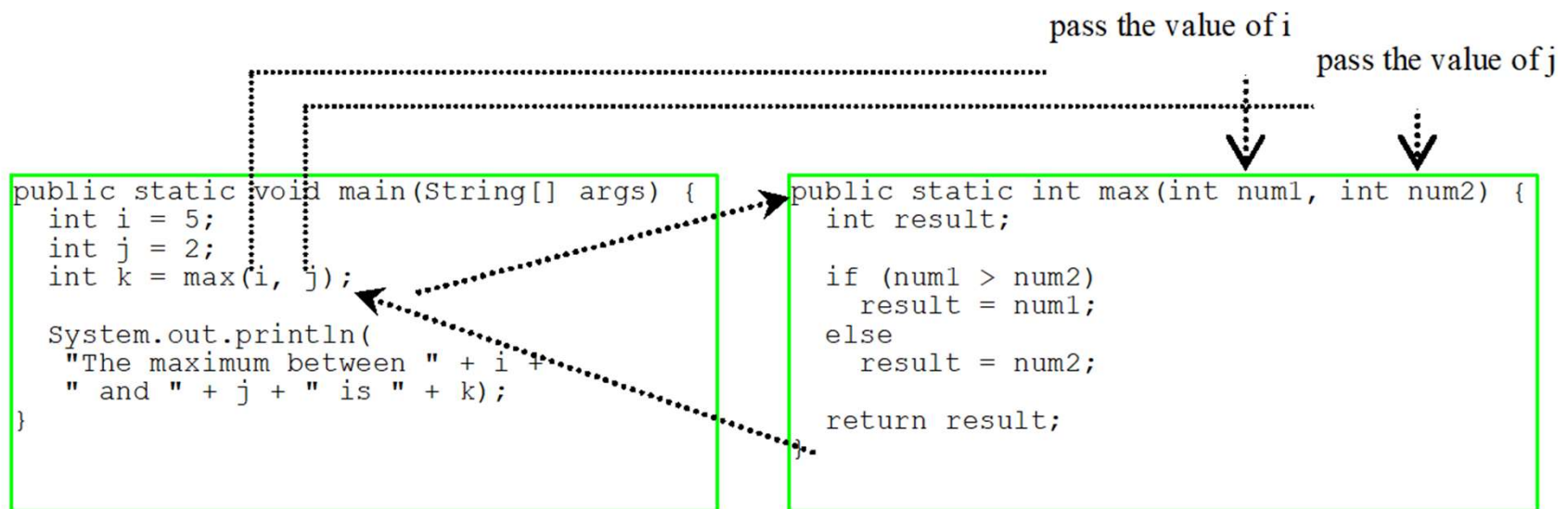
A method may return a value. The **returnValueType** is the data type of the value the method returns. If the method does not return a value, the **returnValueType** is the keyword **void**. For example, the **returnValueType** in the **main** method is **void**.



Calling Methods

Testing the `max` method

This program demonstrates calling a method `max` to return the largest of the `int` values

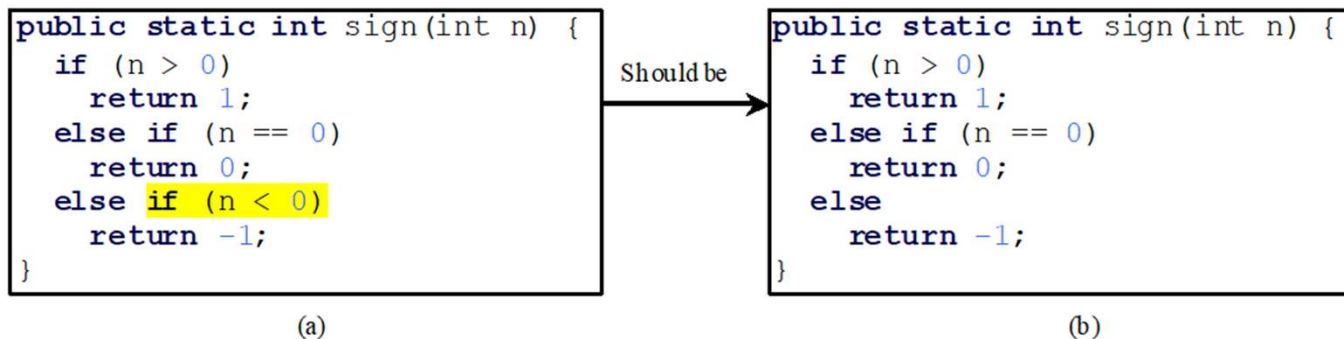


TestMax

TestMax (4 integers)

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.



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 **max** method using **ClassName.methodName** (e.g., **TestMax.max**).

Opening Problem

```
int sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;
```

```
System.out.println("Sum from 1 to 10 is " + sum);
```

```
sum = 0;  
for (int i = 20; i <= 30; i++)  
    sum += i;
```

```
System.out.println("Sum from 20 to 30 is " + sum);
```

```
sum = 0;  
for (int i = 35; i <= 45; i++)  
    sum += i;
```

```
System.out.println("Sum from 35 to 45 is " + sum);
```

[SumDemo](#)

void Method Example

This type of method does not return a value. The method performs some actions.

Given the test score, find the Letter grade

90-100 → 'A'

80 – 89 → 'B'

70 – 79 → 'C'

60 – 69 → 'D'

Below 60 → 'F'

[TestVoidMethod](#)

[TestReturnGradeMethod](#)

Passing Parameters

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

Suppose you invoke the method using

```
nPrintln("Welcome to Java", 5);
```

What is the output?

Suppose you invoke the method using

```
nPrintln("Computer Science", 15);
```

What is the output?

Can you invoke the method using

```
nPrintln(15, "Computer Science");
```

Pass by Value (1 of 3)

This program demonstrates passing values to the methods.

```
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
10 public static void increment(int x) {
11     x++;
12     System.out.println("n inside the method is " + x);
13 }
14 }
```

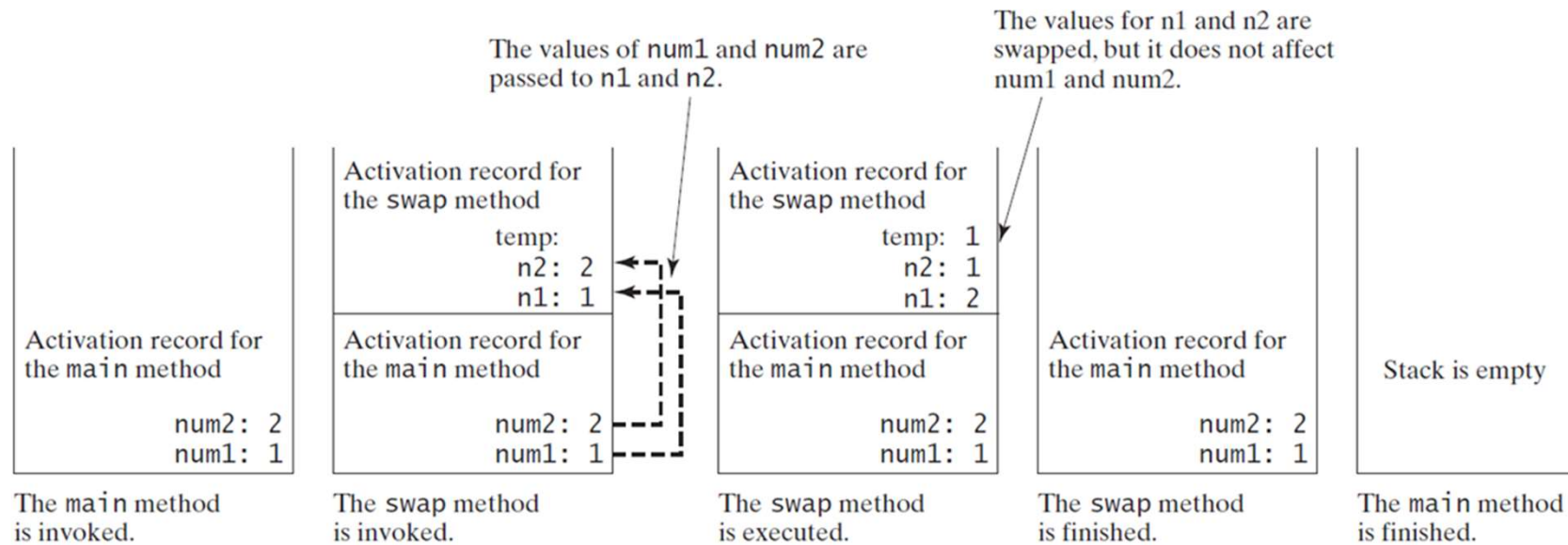
Increment

Pass by Value (2 of 3)

```
1 // Given two int values, swap them
2 public class TestPassByValue {
3     public static void main(String[] args){
4         int num1 = 5;
5         int num2 = 10;
6         System.out.println("Before swap()");
7         System.out.println("X is " + num1 + " and Y is " + num2);
8         swap(num1,num2);
9         System.out.println("After swap()");
10        System.out.println("X is " + num1 + " and Y is " + num2);
11    }
12
13    public static void swap(int n1,int n2){
14        int temp = n1;
15        n1 = n2;
16        n2= temp;
17    }
18 }
19
```

TestPassByValue

Pass by Value (3 of 3)



Modularizing Code

GCD V2

Methods can be used to reduce redundant coding and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

```
1 import java.util.*;
2 public class GCD {
3     public static void main(String[] args) {
4         Scanner in = new Scanner(System.in);
5         int n1 = in.nextInt();
6         int n2 = in.nextInt();
7         int gcd = 1;
8         int k = Math.min(n1,n2);
9         while (k>=2) {
10             if(n1%k==0 && n2%k==0) {
11                 gcd = k;
12                 break;
13             }
14             k--;
15         }
16         System.out.println("GCD is "+gcd);
17     }
18 }
```

Modularizing Code

PrimeNumberMethod

Methods can be used to reduce redundant coding and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

```
1 import java.util.*;
2 public class PrimeNumbers {
3     public static void main(String[] args) {
4         Scanner in = new Scanner(System.in);
5         int count = 0;
6         int num = 2;
7         while(count != 50){
8             boolean isPrime = true;
9             for(int i=2;i<num;++i) {
10                 if(num % i == 0) {
11                     isPrime = false;
12                     break;
13                 }
14             }
15             if(isPrime){
16                 count++;
17                 if(count%10!=0)
18                     System.out.print(num+" ");
19                 else
20                     System.out.println(num);
21             }
22             num++;
23         }
24     }
25 }
26 }
27 }
```

Case Study: Converting Hexadecimals to Decimals

Write a method that converts a hexadecimal number into a decimal number.

ABCD \Rightarrow

$$A * 16^3 + B * 16^2 + C * 16^1 + D * 16^0$$

$$= ((A * 16 + B) * 16 + C) * 16 + D$$

$$= ((10 * 16 + 11) * 16 + 12) * 16 + 13 = ?$$

[Hex2Dec](#)

Overloading Methods

Overloading the `max` Method

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

[TestMethodOverloading](#)

Ambiguous Invocation (1 of 2)

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 (2 of 2)

```
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;  
    }  
}
```

Scope of Local Variables (1 of 6)

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 (2 of 6)

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 (3 of 6)

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 →

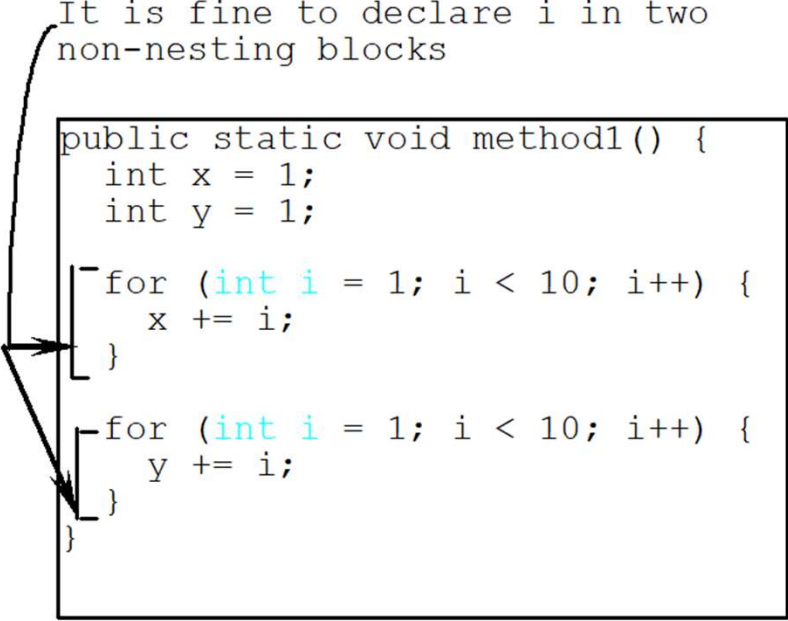
The scope of j →



Scope of Local Variables (4 of 6)

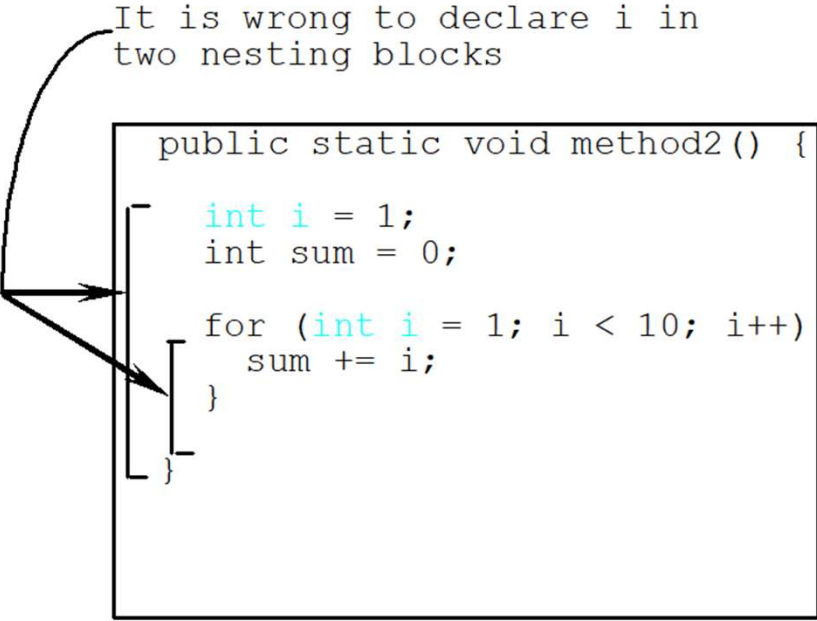
It is fine to declare `i` in two non-nesting blocks

```
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 (5 of 6)

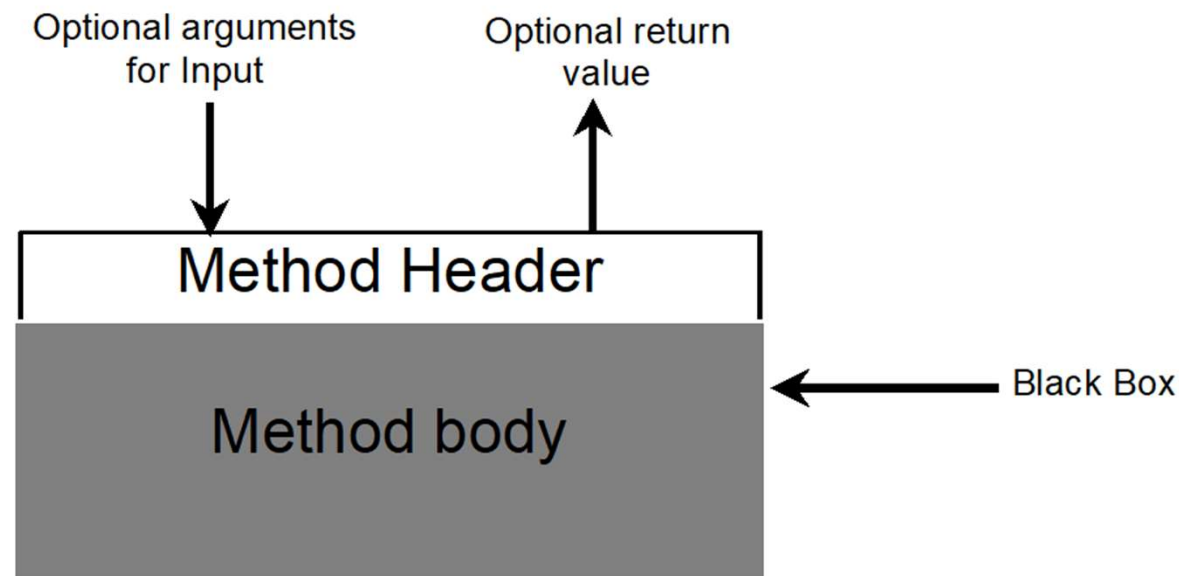
```
// Fine with no errors
public static void correctMethod() {
    int x = 1;
    int y = 1;
    // i is declared
    for (int i = 1; i < 10; i++) {
        x += i;
    }
    // i is declared again
    for (int i = 1; i < 10; i++) {
        y += i;
    }
}
```

Scope of Local Variables (6 of 6)

```
// 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.