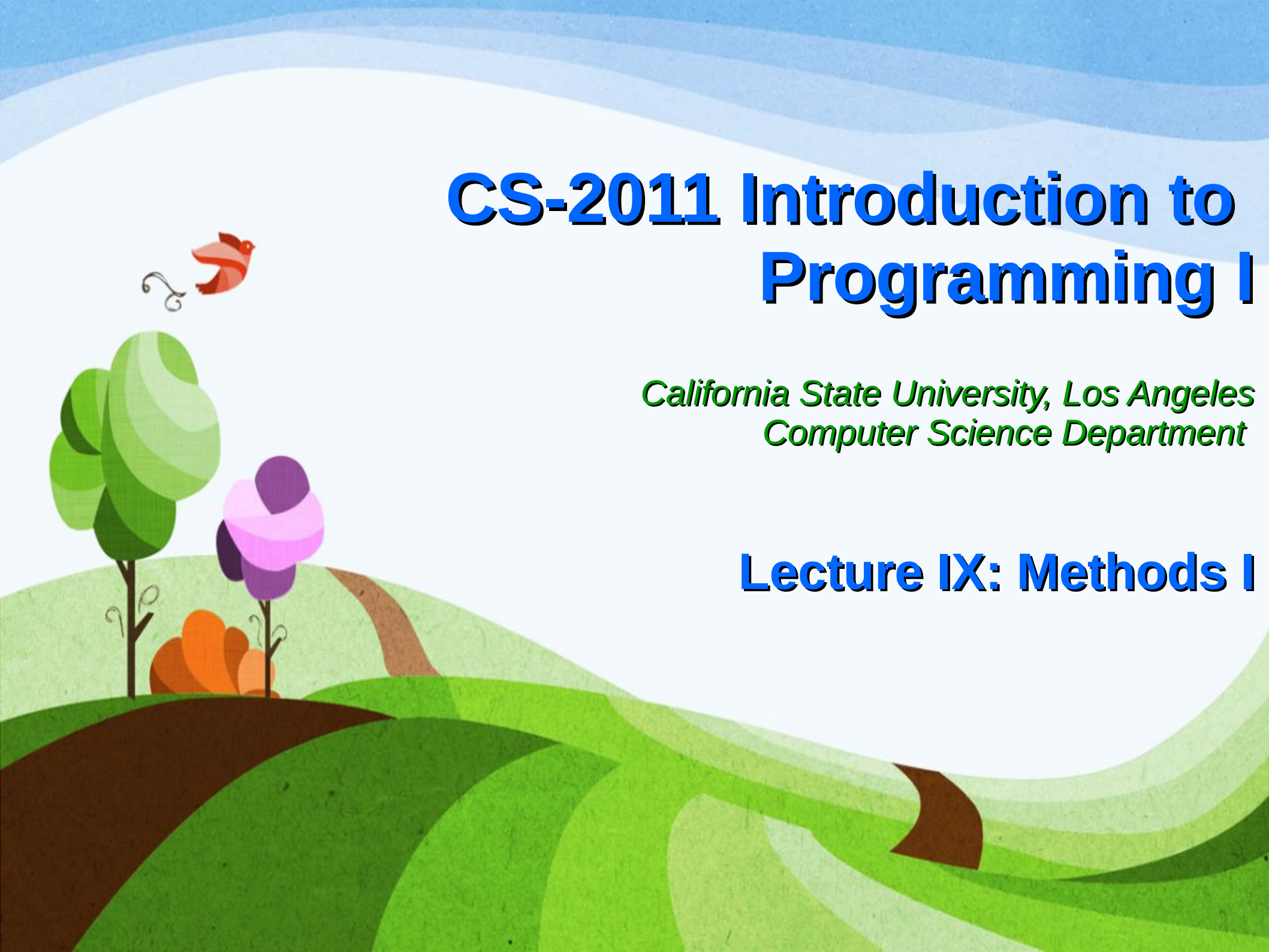


CS-2011 Introduction to Programming I

California State University, Los Angeles
Computer Science Department

Lecture IX: Methods I

Introduction

- **method**:
  - construct for grouping statements together to perform a function
  - write a piece of code one time and reuse over and over
  - can be reused anywhere in the same program or even anywhere in different programs
  - in other programming languages methods are called:
    - functions
    - procedures
    - subroutines

Introduction

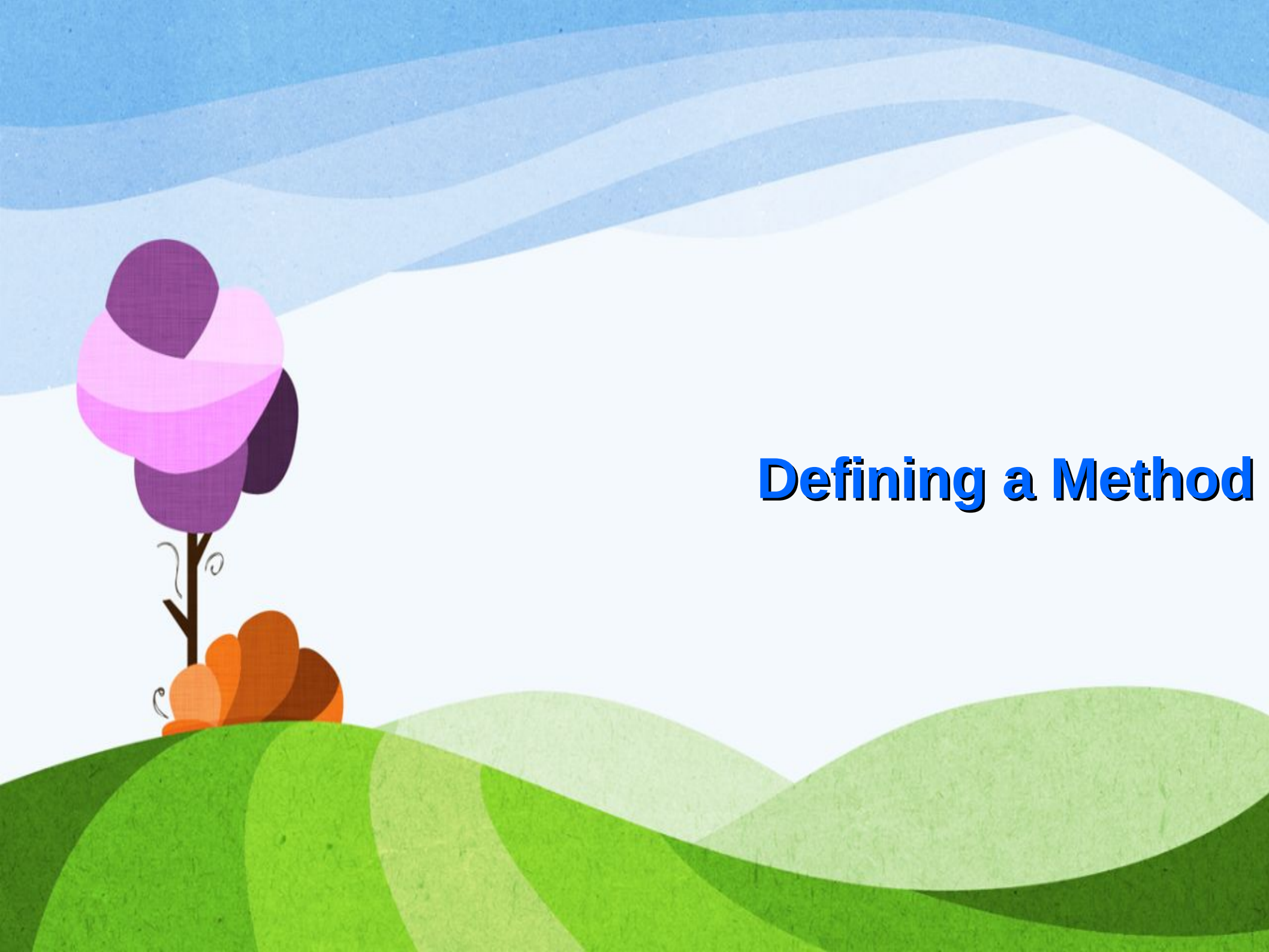
- Find the sum of integers from 1 to 10, 20 to 37, and 35 to 49.
- The naïve approach would be this:

```
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 <= 37; i++)
  sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

Introduction

- The preceding code can be simplified:

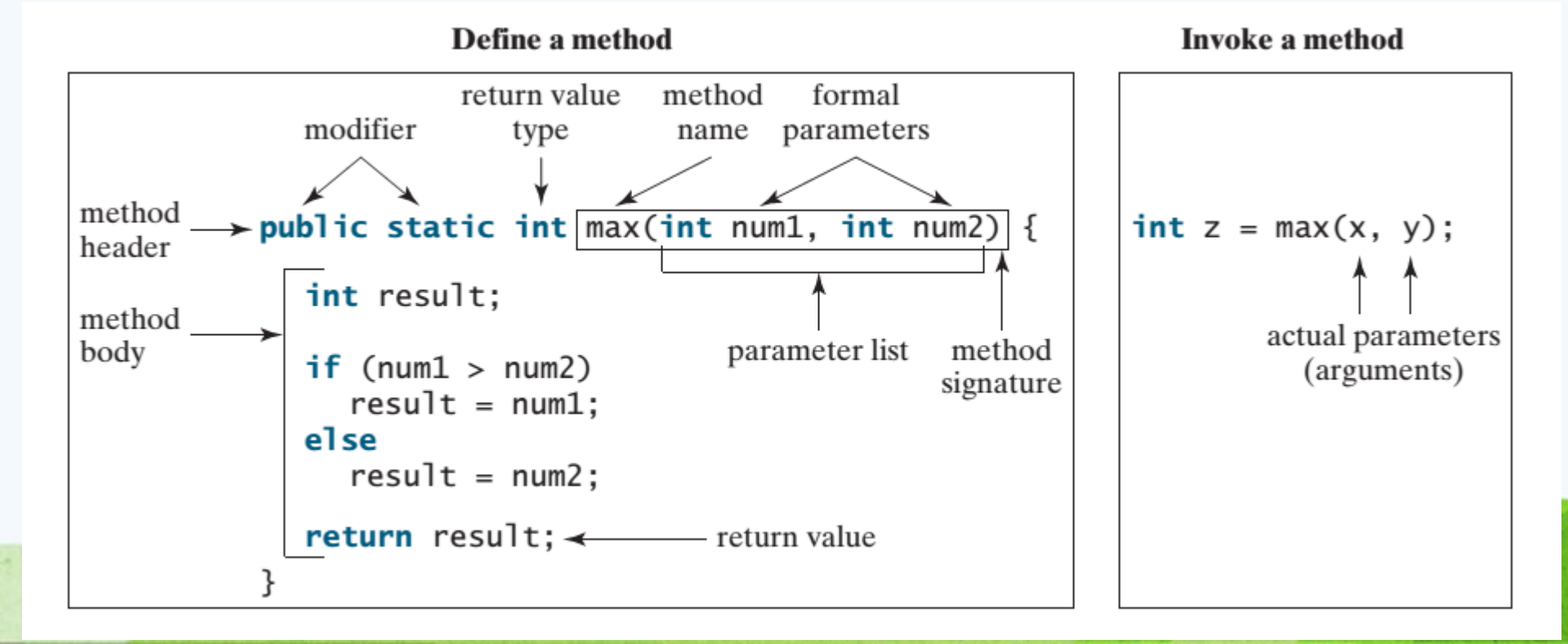
```
1  public static int sum(int i1, int i2) {
2    int result = 0;
3    for (int i = i1; i <= i2; i++)
4      result += i;
5
6    return result;
7  }
8
9  public static void main(String[] args) {
10   System.out.println("Sum from 1 to 10 is " + sum(1, 10));
11   System.out.println("Sum from 20 to 37 is " + sum(20, 37));
12   System.out.println("Sum from 35 to 49 is " + sum(35, 49));
13 }
```


Defining a Method

Defining a Method

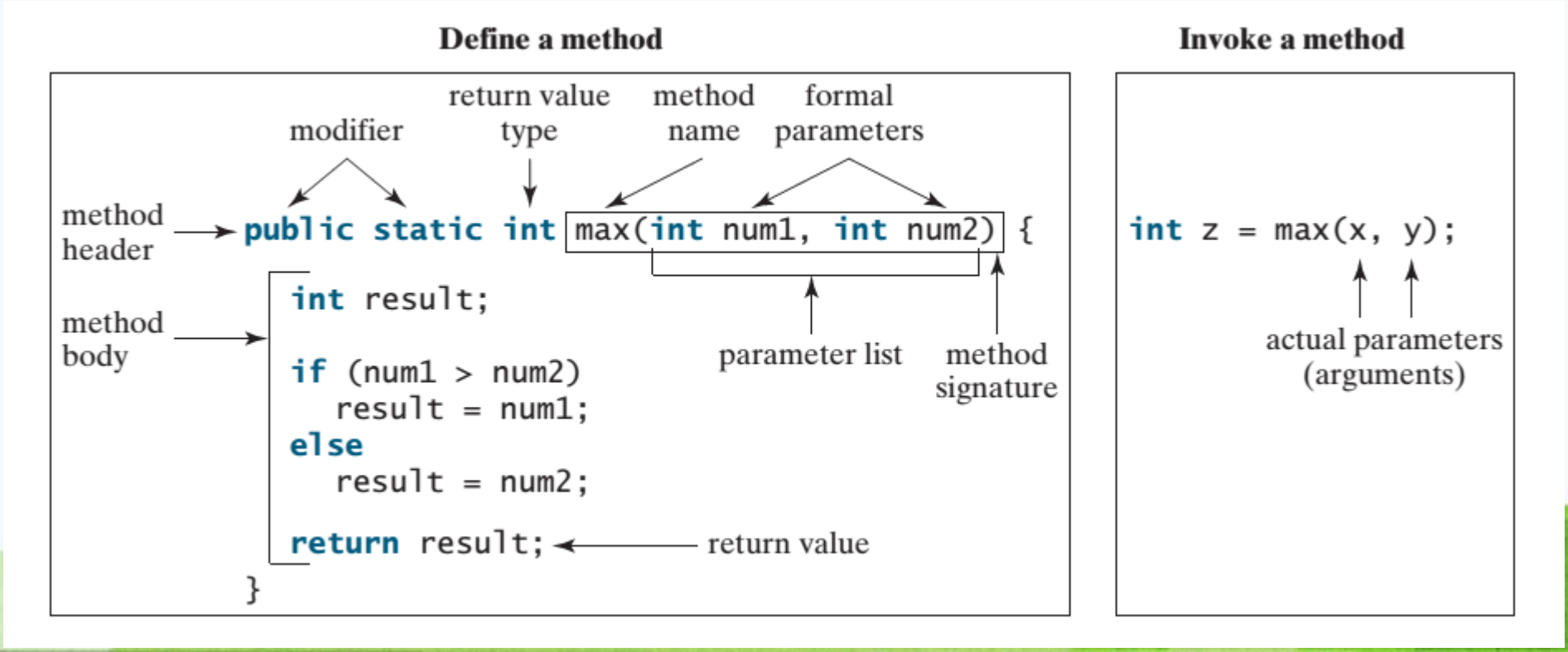
- ***method definition*:** the method name, parameters, return value type, and body.
- Syntax:

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



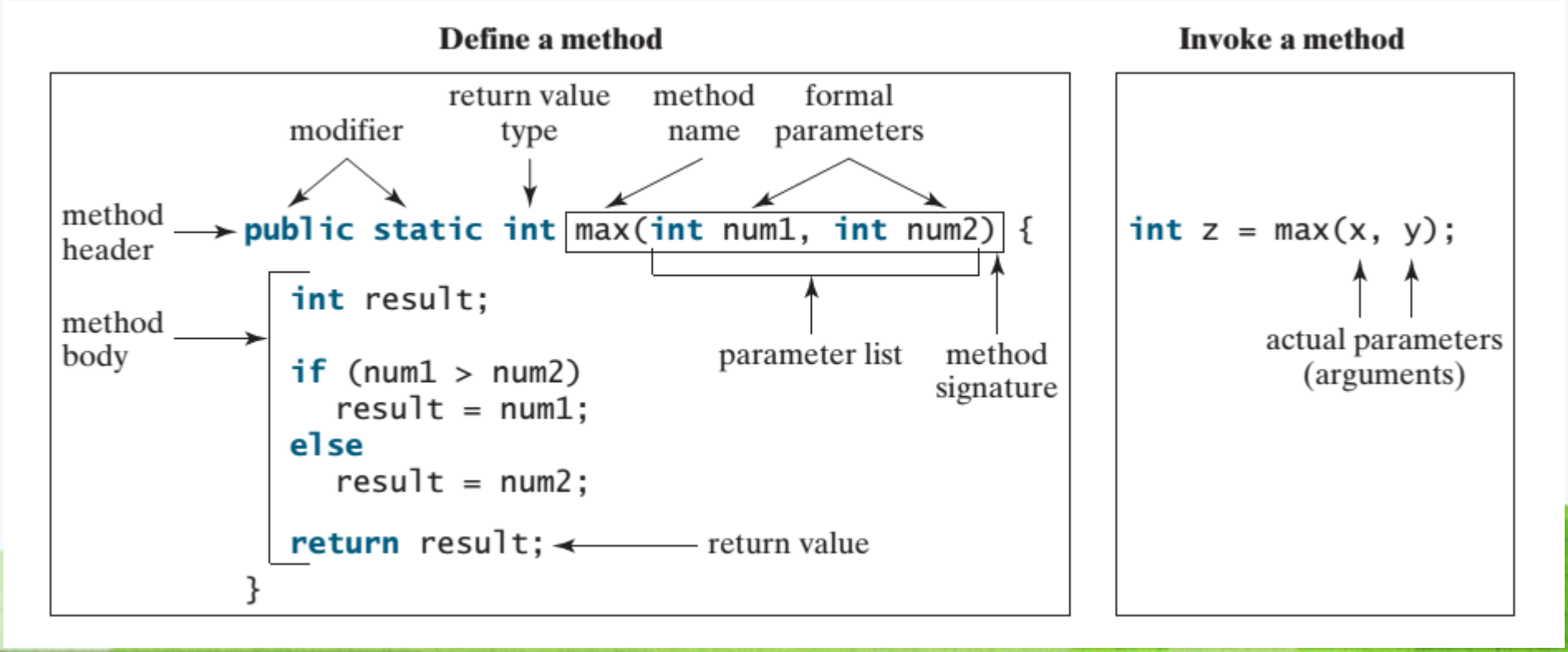
Method Signature

- ***method signature*: the method name and the parameter list make up the **signature of the method**.**



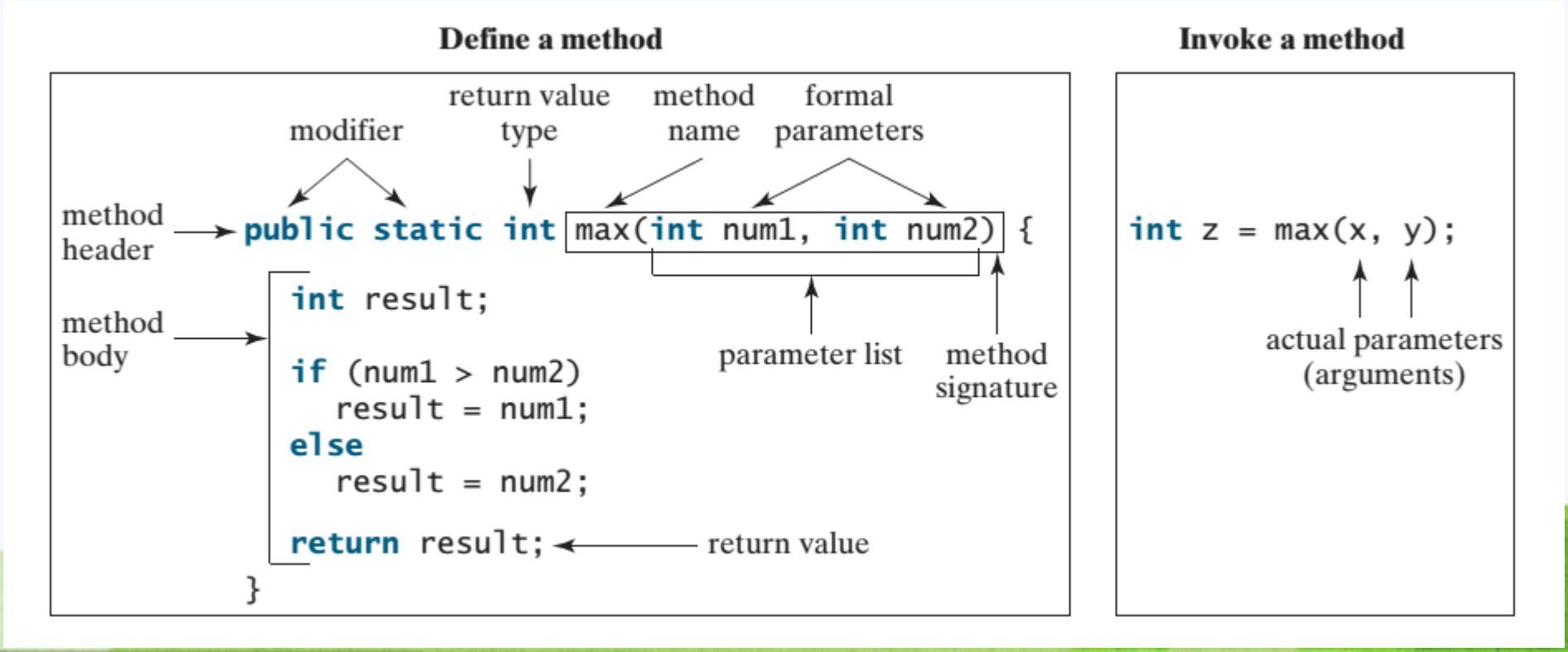
Formal Parameters (Parameters)

- **formal parameters (or parameters):** local variables defined in the method signature
  - refers to the data type of the parameters, order in which they appear, and number of parameters.
  - are completely optional.
  - each parameter needs its own data type.
    - `max(int num1, int num2)  //correct`
    - `max(int num1, num2)  //incorrect`
  - Think of formal parameters like a placeholder for a value the method will accept in the future when the method is called.



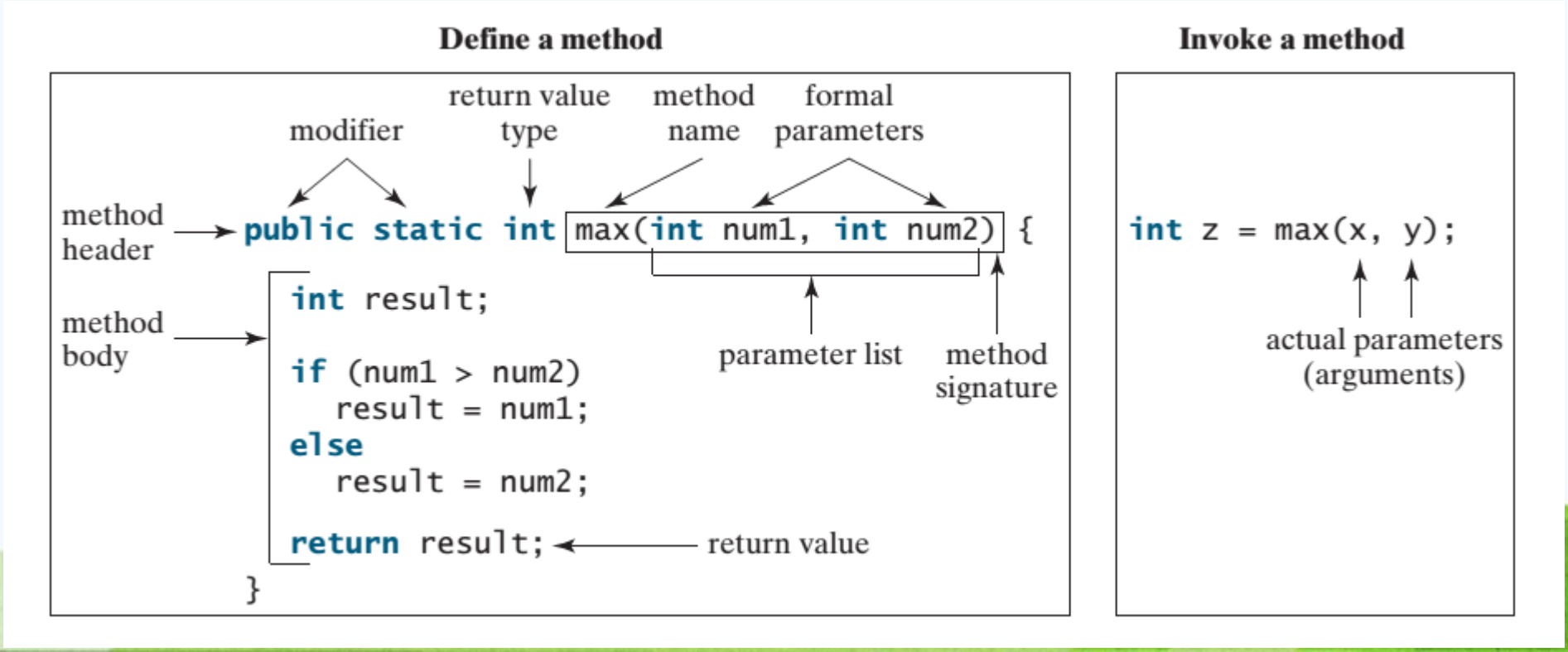
Actual Parameters (Arguments)

- When a method is invoked you must pass a value to the parameter.
- This value is the ***actual parameter (or argument)*** of the method.



Return Value Type

- A method can return a value and the ***return value type*** is the data type of the value the method returns.
- Methods that do not return a value use the ***void*** keyword as the ***return value type***. An example is the main method.




Calling a Method

Calling a Method

- method definition - defines what the method should do.
  - does not execute the code.

- to actually use a method, you must call or invoke it.

- **invoking a method (method invocation):** the act of calling a method in your code:
  - Math.pow() //calls the pow() method of the Math class
  - in.nextInt() //calls the nextInt() method of the Scanner class
  - same concept when you invoke a method you write.

Calling a Method

- Value Returning Method
  - invoking this type of method is usually treated as a value
  - `int largerNumber = max(3, 4);`
  - `System.out.println(max(3, 4));`

- Void Method (Method does not return a value)
  - invoking this type of method must be a statement
  - cannot be assigned to a variable.
  - `System.out.println("Welcome to Java");`

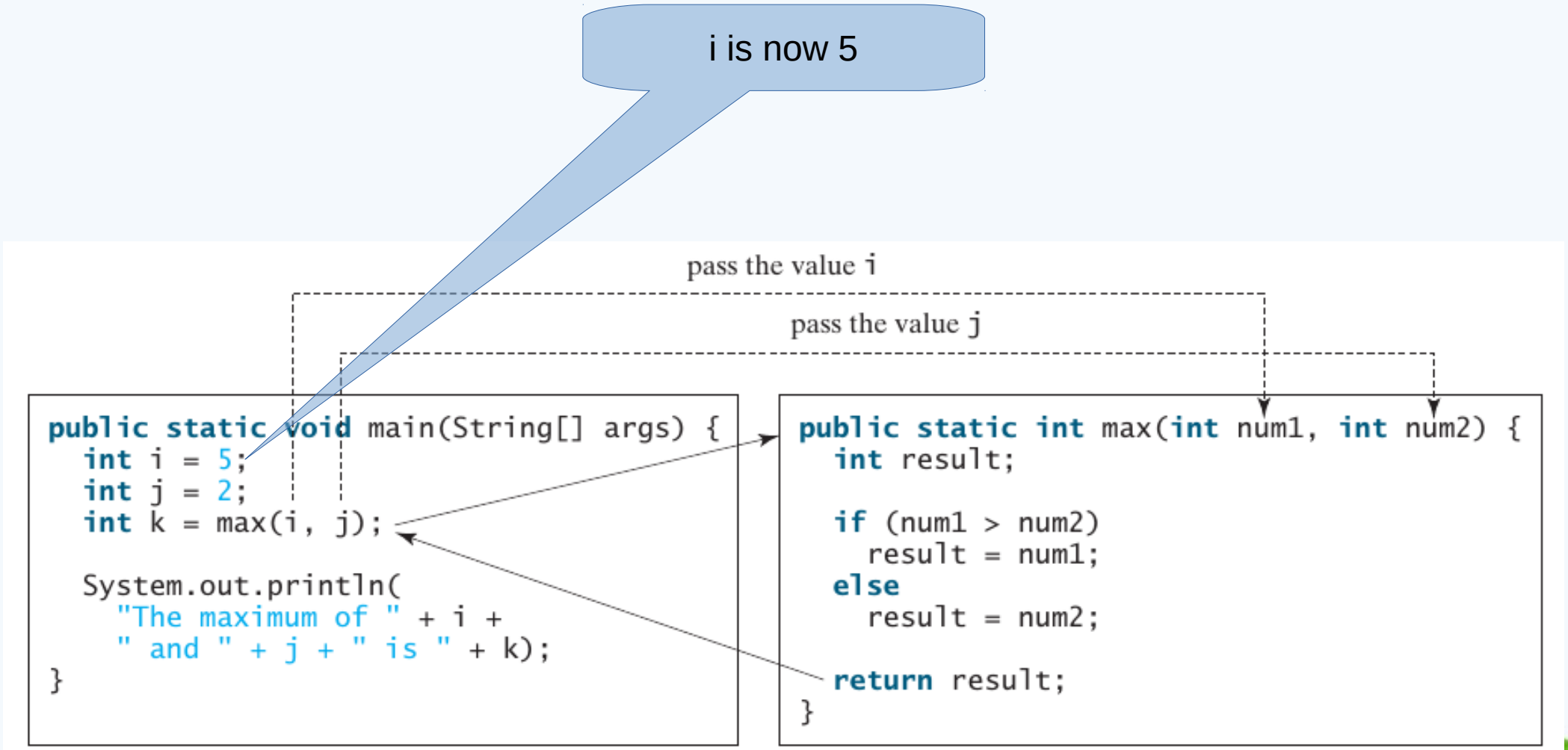
Calling a Method

- When a method is called, program control is transferred to the called method

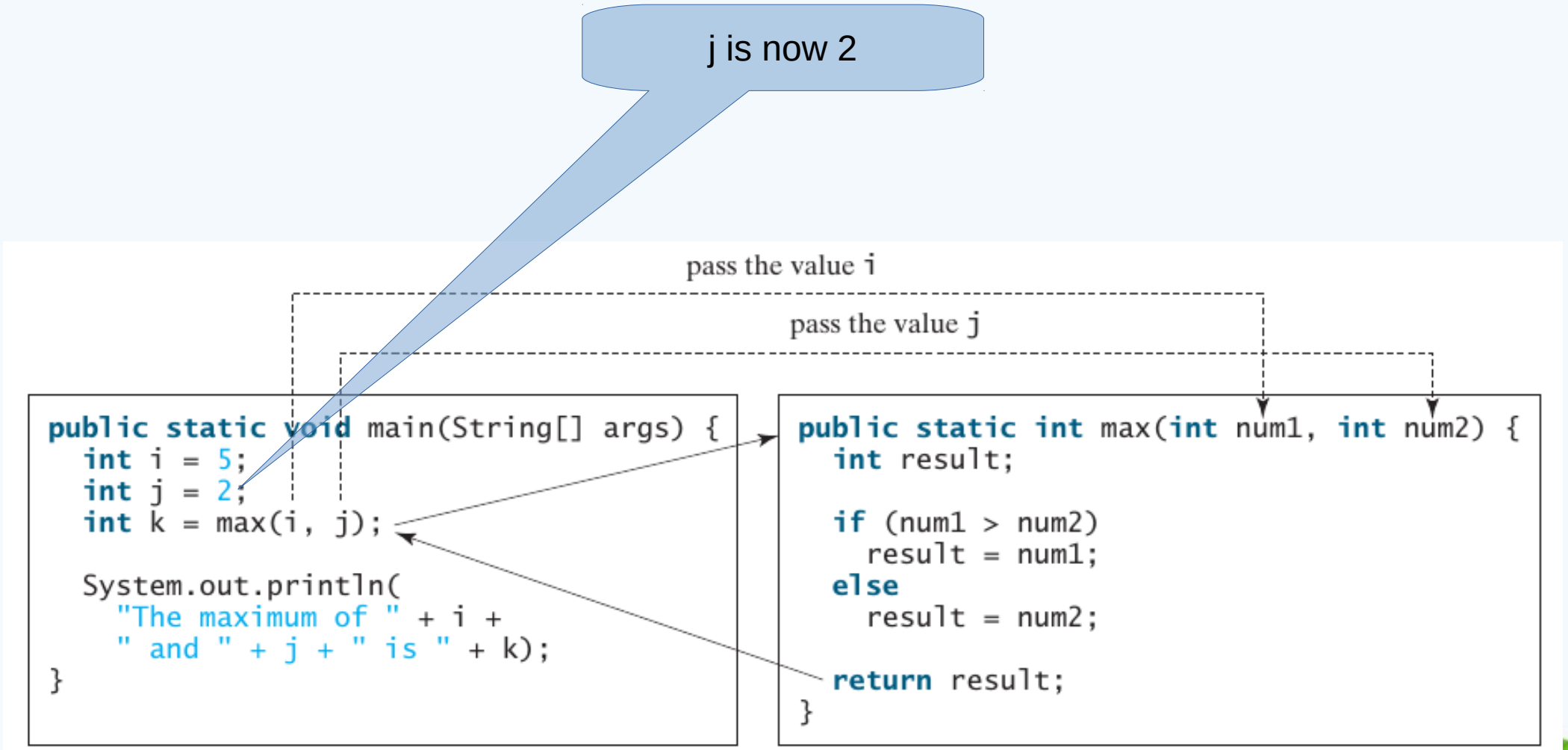
- Control is transferred back to the caller when:
  - the called method's statement is executed
  - or its method ending closing brace is reached

- See Code: **`TestMax.java`**

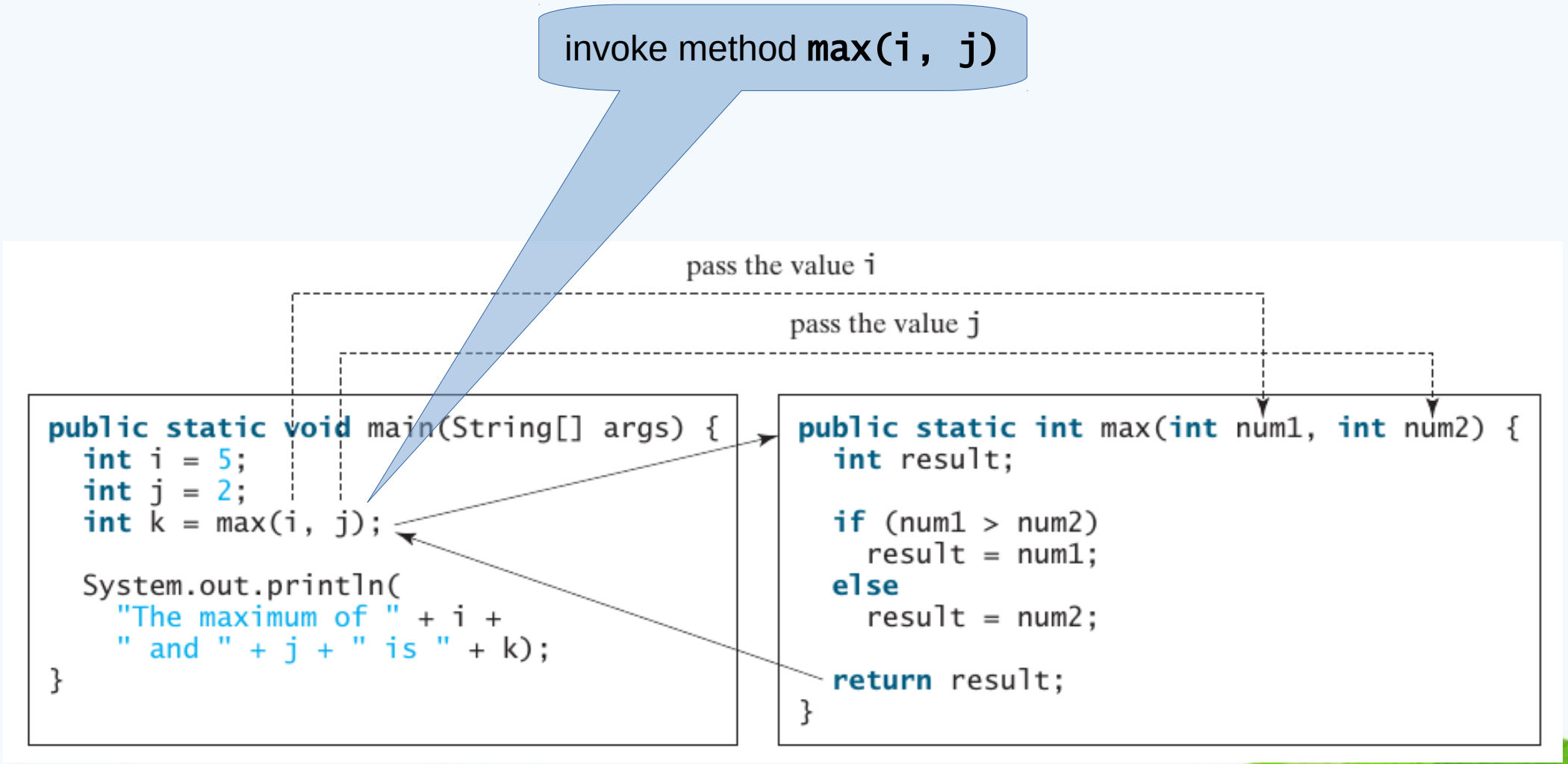
TestMax.java Trace



TestMax.java Trace

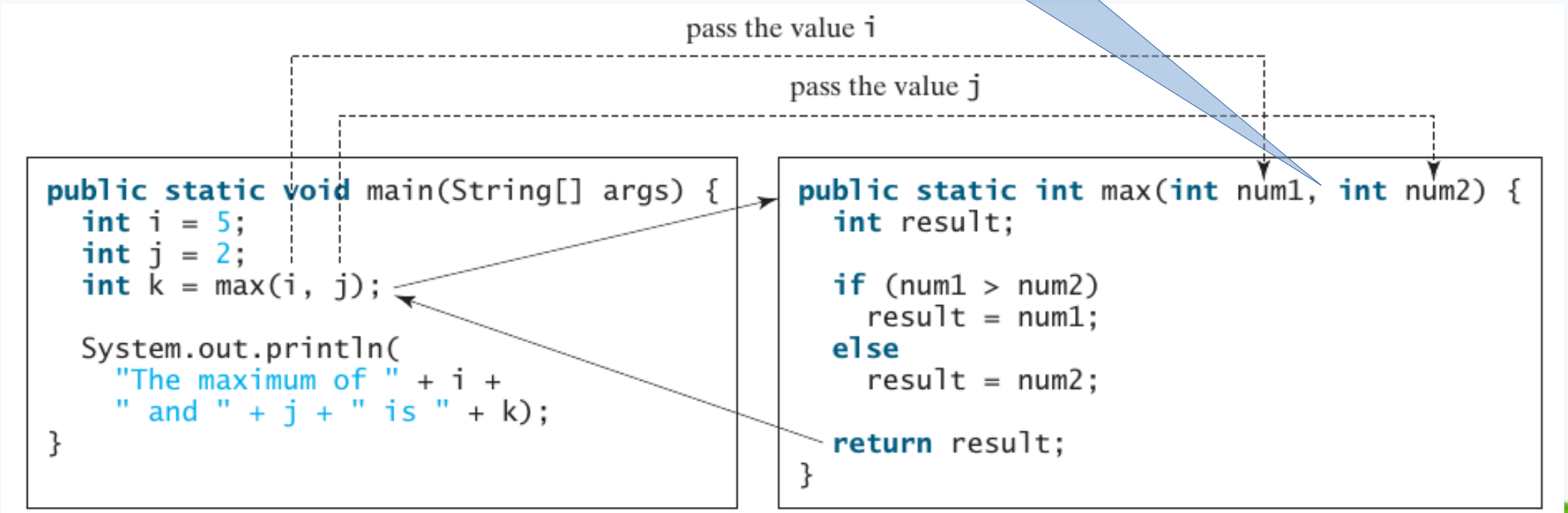


TestMax.java Trace

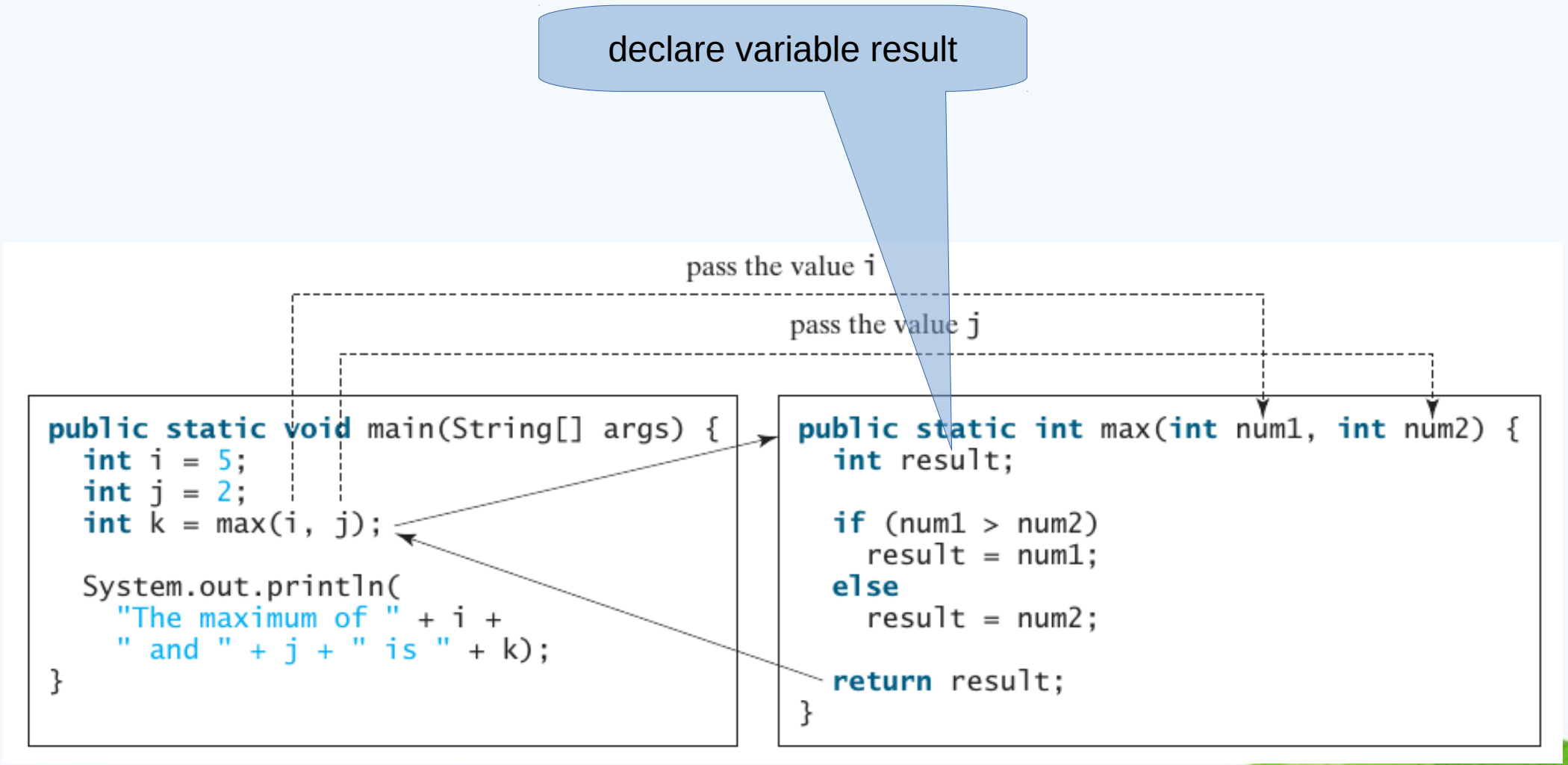


TestMax.java Trace

invoke **max(i, j)**
Pass the **value** of i to num1
Pass the **value** of j to num2

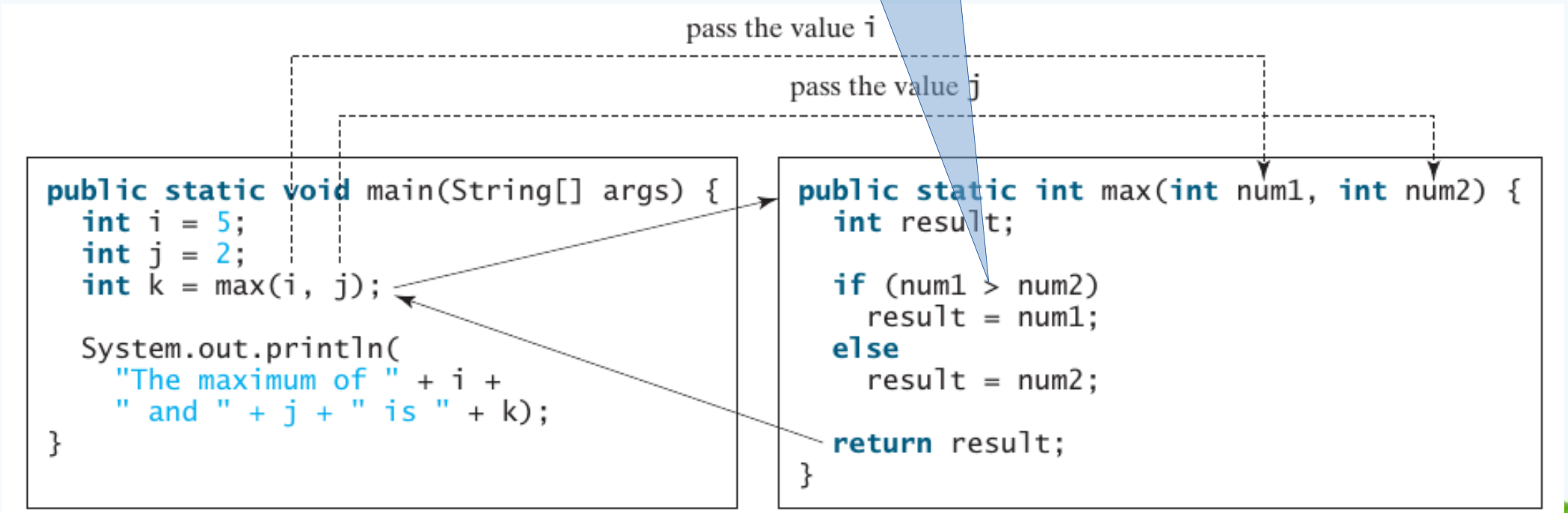


TestMax.java Trace

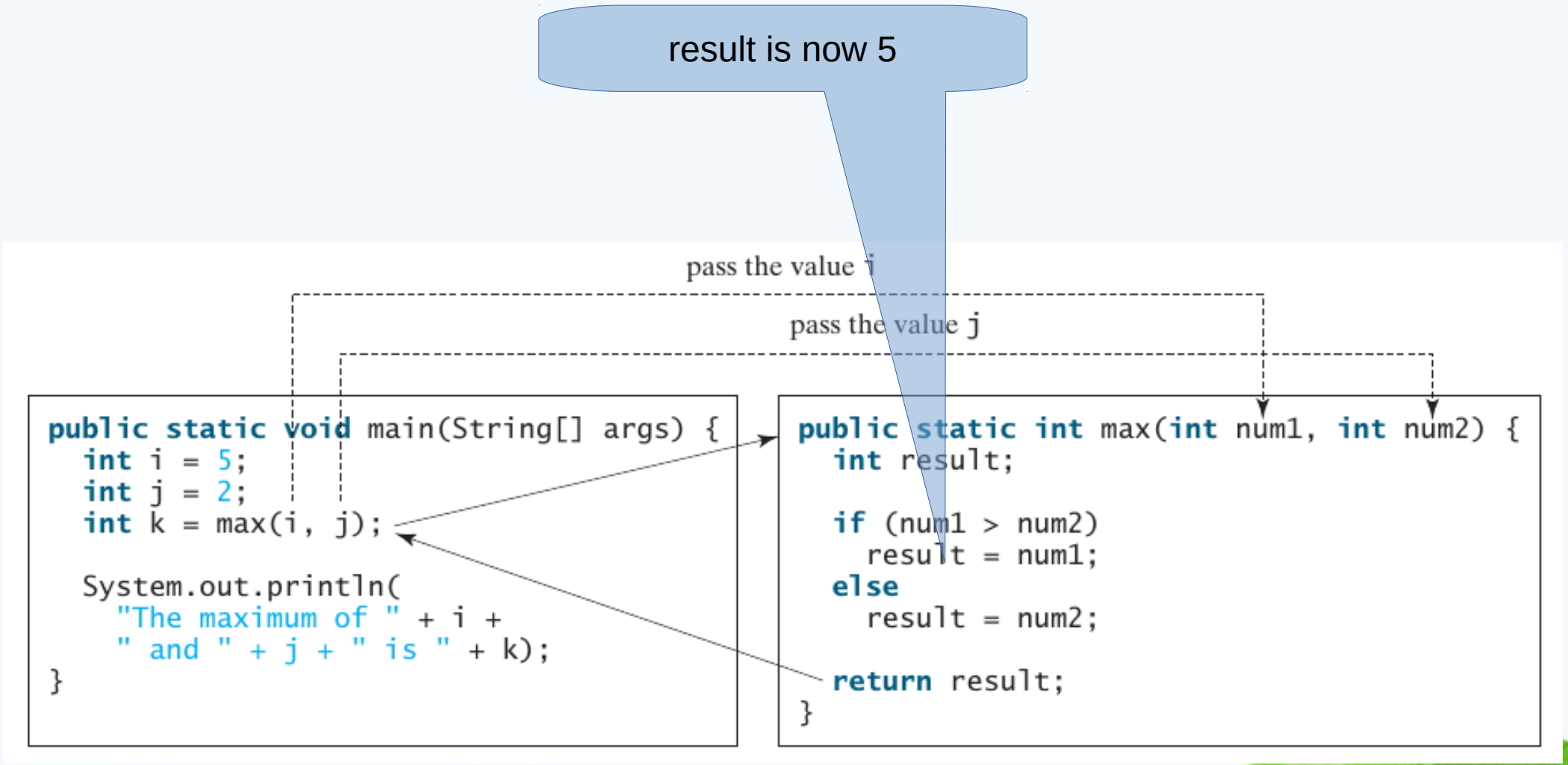


TestMax.java Trace

(num1 > num2) is true since num1 is 5 and num2 is 2

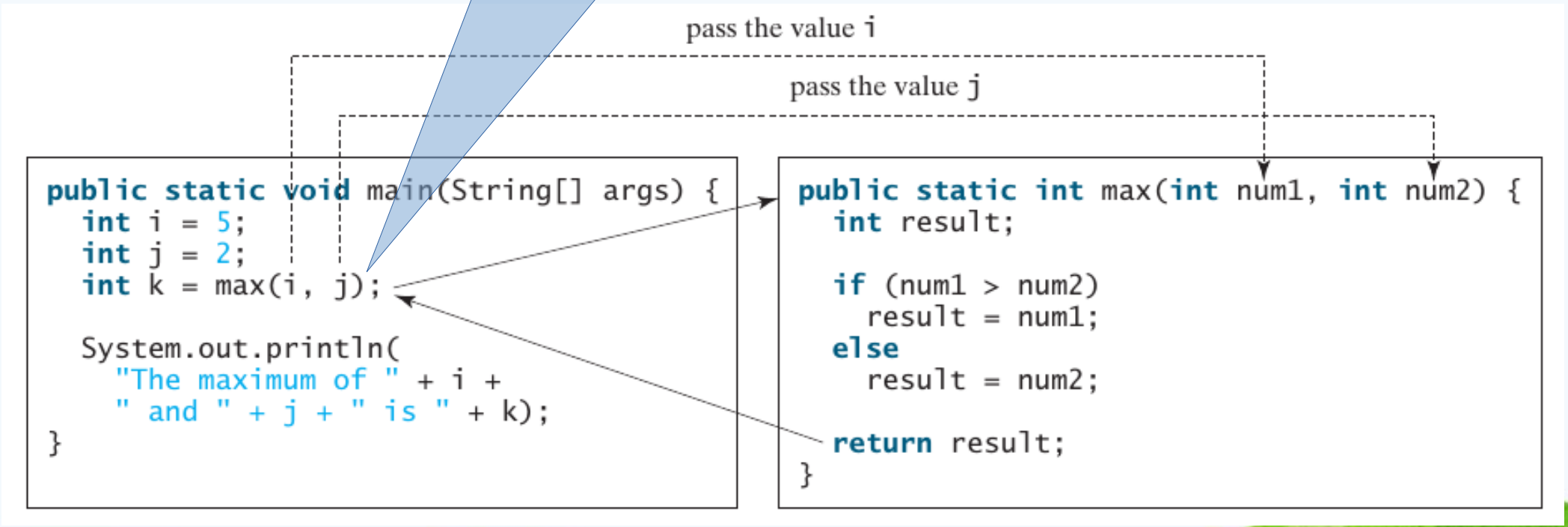


TestMax.java Trace

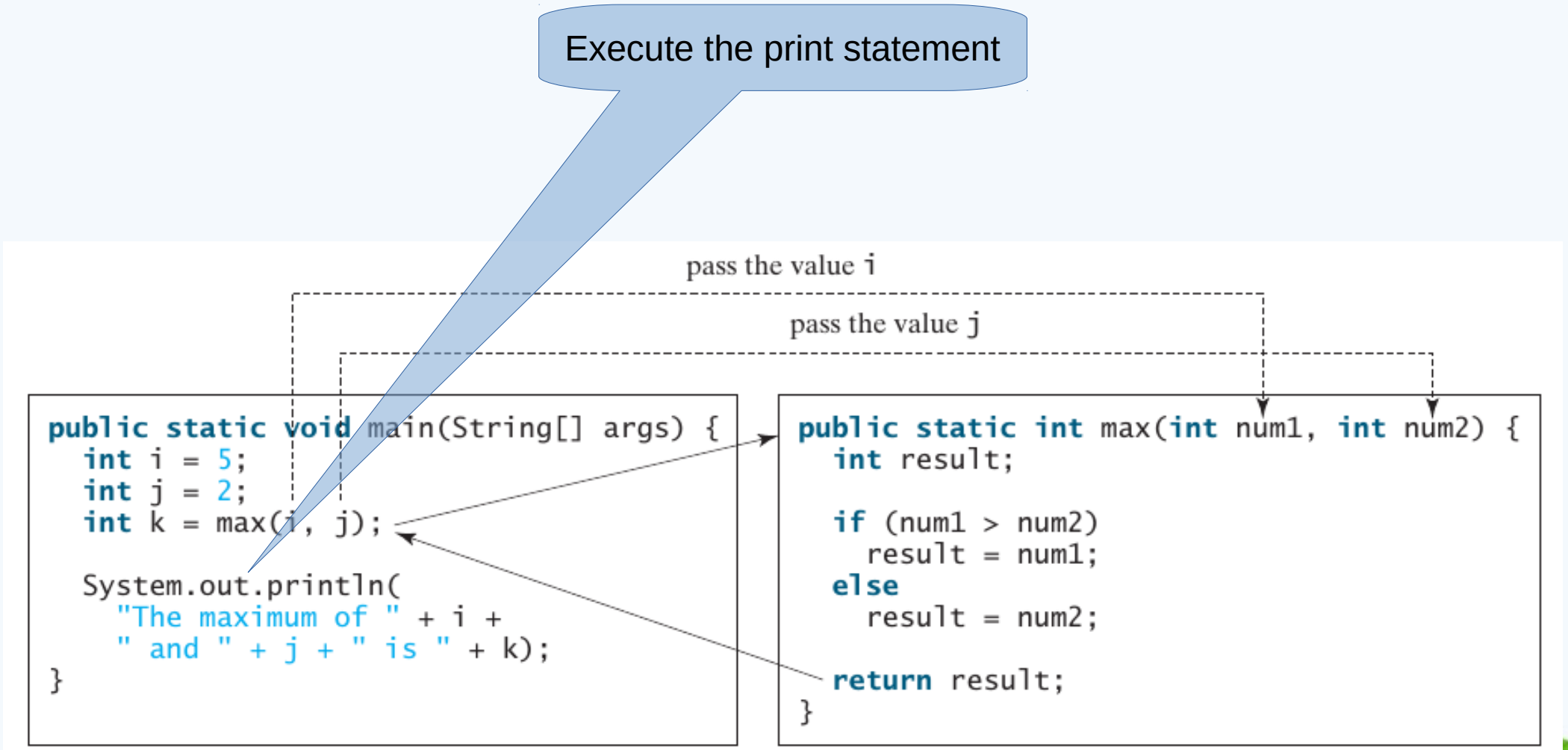


TestMax.java Trace

return to where **`max(i, j)`** was called
`max(i, j)` assigns the returned value to k



TestMax.java Trace



A Word of Caution

- A ***return*** statement is required for **methods that return a value**.
- example below is logically correct, but there is a compilation error.
  - compiler thinks it is possible that the method does not return any value since all cases of the if/else might be false.

```
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 can be reached regardless of how the ***if*** statement is evaluated

Reusing Methods from Other Classes

- methods provide a way to reuse code

- Methods that have the **`public`** and **`static`** modifiers can be invoked using

```
Classname.methodName(arguments)
```

```
TestMax.max(x, y);
```

  - same way you call methods from the Math class.


void Methods

`void` Methods

- A **`void`** method does not return a value it just performs some action.
  - has a return type of **`void`**.

- A return statement is not required, but it can be used for terminate the method early.

- See Code:
  - **`TestVoidMethod.java`**
  - **`TestReturnGradeMethod.java`**


The Scope of Variables

Scope of Variables

- ***local variable***: a variable defined **inside** a method
  - A local variable must be declared before it can be used.

- (in CS-2012 you will learn about class-level variables, and instance variables)

- ***scope***: the part of the program where a variable can be referenced / used.

- scope of a local variable starts from its declaration and ends at the end of the block that contains the variable.
  - method blocks
  - if/else blocks
  - loop blocks

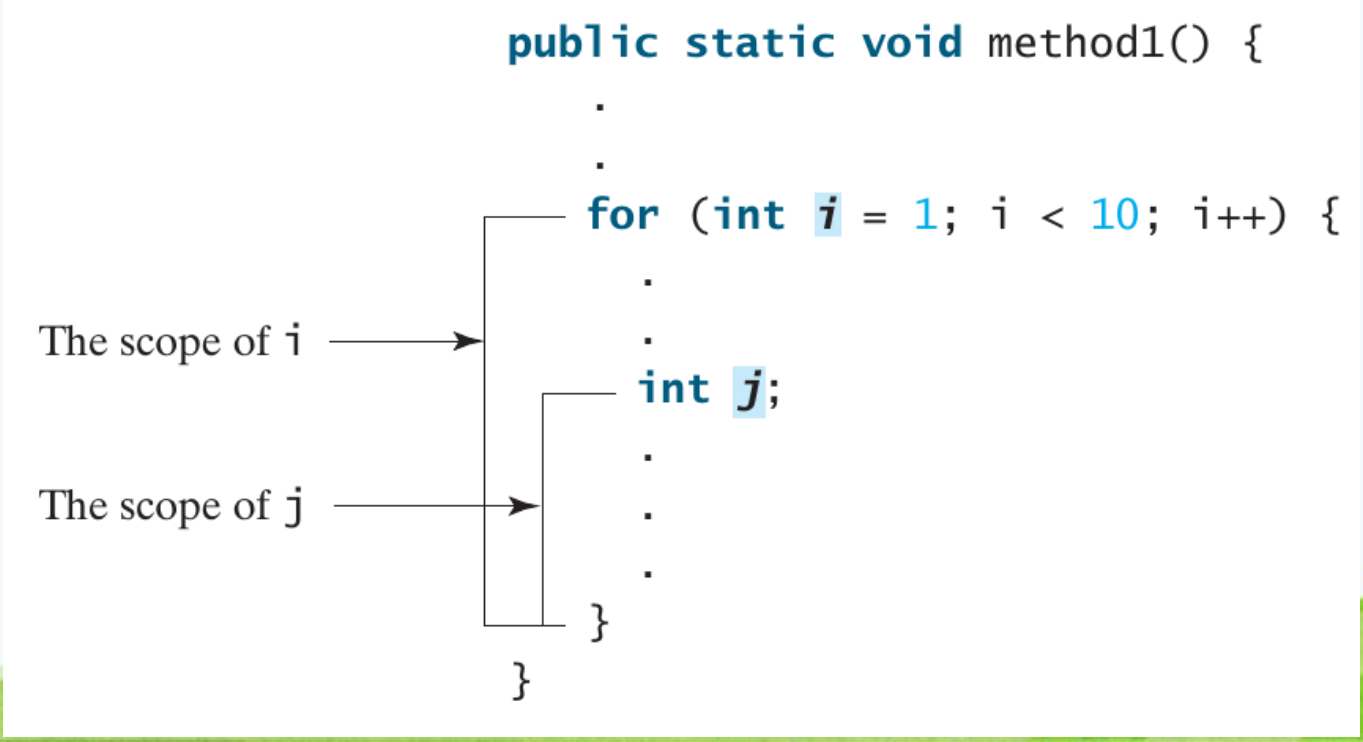
Scope of Variables

- method parameters are **always** local variable.
  - the scope of the parameter is the entire method in which it is declared.
  - the memory allocated for the parameter will be freed up after the method is completed and the variable will no longer be available.

Scope of Variables

- the scope of a variable declared in the initial-action part of a **`for`** loop header is the entire loop.

- the scope of a variable declared inside the body of a for loop starts from its declaration and ends at the closing curly brace of the loop that contains the variable



Scope of Variables

- you can declare a local variable with the same name in different blocks in a method.

- you CANNOT declare a local variable twice in the same block or in nested blocks.

It is fine to declare `i` in two nonnested 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 nested blocks.

```
public static void method2() {

  int i = 1;
  int sum = 0;

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

}
```


Passing Arguments

Passing Arguments

- methods can accept outside information
  - input to the method
  - always provided by the calling method
  - makes methods reusable since you can use the same method with different sets of data.

- outside information is passed as **arguments** to the **parameters** of the method.

Passing Arguments

- Recall:
  - **arguments** are the **actual** values passed in the method call.
  - **parameters** are the place holder variables in the method definition.

- Example: if I have a method with a header of:

```
public static void myMethod(int x, double y, String s)
```

and I call the method using `myMethod(5, 12.6, "Java");`

`x`, `y`, and `s` are the parameters
`5`, `12.6`, and `"Java"` are the arguments passed to the parameters

Passing Arguments

- If a method header has parameters:
  - you **MUST** pass arguments to the method call.
- if a method has no parameters:
  - you **CANNOT** pass arguments to the method call.
  - you **MUST** still include an empty set of parenthesis.
- Example:
  - **`Math.pow(a, b)`** is method where you are required to pass two arguments, a base and an exponent.
  - **`nextInt()`** of the Scanner class is a method were you CANNOT pass arguments, but the empty ( ) are still required.

Passing Arguments

- ***parameter order association:*** arguments passed to parameters **MUST:**
  - be passed in the correct order
  - have the exact number of arguments (no more or no less than what the method defines)
  - match the parameters based on compatible types

- Compatible type:
  - you can pass an argument to a parameter without explicit casting
  - i.e. passing an int value to a double value parameter is ok, but the reverse would not be.

Passing Arguments

```
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?

Passing Values

- a copy of the VALUE of the argument is passed to the parameter:
  - not the actual variable...but the VALUE
  - changes made to a value inside a method will not affect the value outside of the method.
  - **primitives pass their values.**

- Code Examples:
  - **`Increment.java`**
  - **`TestPassByValue.java`**

Passing Arguments by Values