COMP30820
Java Programming (Conv)

Michael O'Mahony

Chapter 6 Methods

Opening Example

Consider the following problems:

- Find the sum of integers from 1 to 10, inclusive.
- Find the sum of integers from 20 to 30, inclusive.
- Find the sum of integers from 35 to 45, inclusive.

These are similar problems (finding the sum), but the ranges differ…

Opening Example

```
public class TestSum {
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 1; i <= 10; i++)
      sum += i;
    System.out.println("Sum 1 to 10 is " + sum);

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

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

Opening Example

```
public class TestSum {
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 1; i <= 10; i++)
      sum += i;
    System.out.println("Sum 1 to 10 is " + sum);

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

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

Solution

```
public class TestSum2 {
  public static void main(String[] args) {
    System.out.println("Sum 1 to 10 is " + sum(1, 10));
    System.out.println("Sum 20 to 30 is " + sum(20, 30));
    System.out.println("Sum 35 to 45 is " + sum(35, 45));
  }

  public static int sum(int n1, int n2) {
    int sum = 0;
    for (int i = n1; i <= n2; i++)
      sum += i;
    return sum;
  }
}
```

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 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).

Defining and Calling Methods

Example program `TestMax` – demonstrates defining and calling a method `max` to return the largest of two `int` values

TestMax

The `TestMax` Class

```
public class TestMax {
   public static void main(String[] args) {
      int i = 5;
      int j = 2;
      int k = max(i, j);

      System.out.println("The max is " + k);
   }

   public static int max(int num1, int num2) {
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
}
```

animation

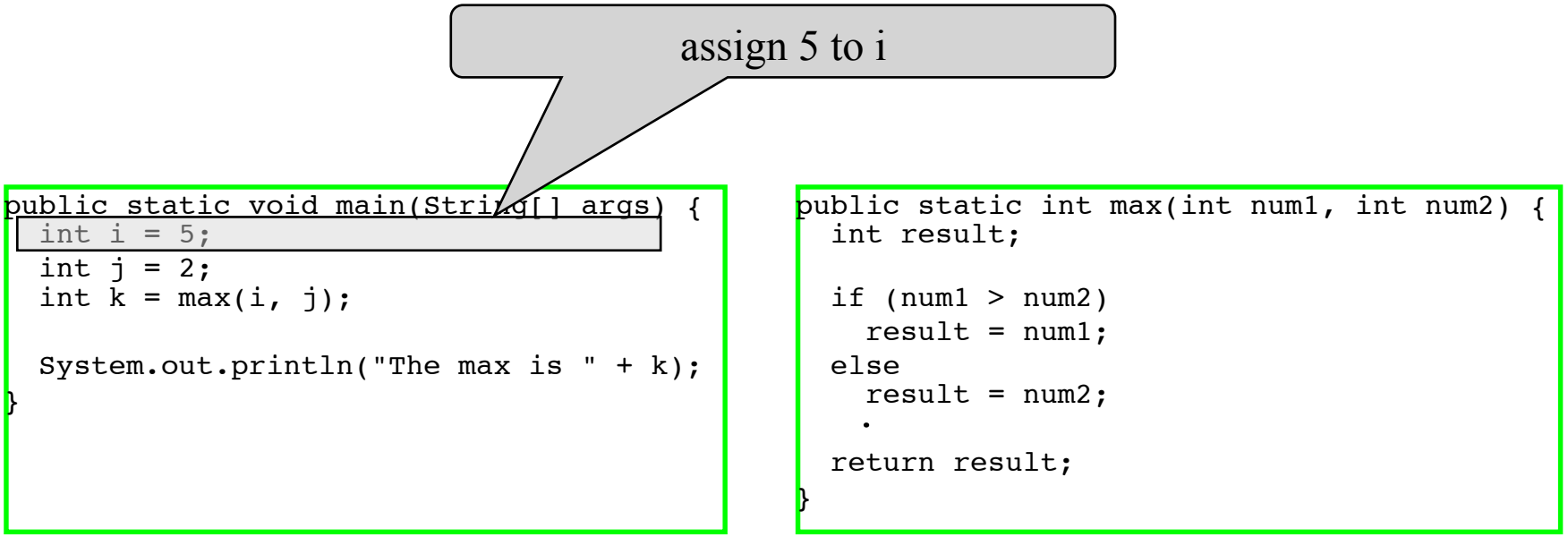
Trace Method Invocation

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println("The max is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

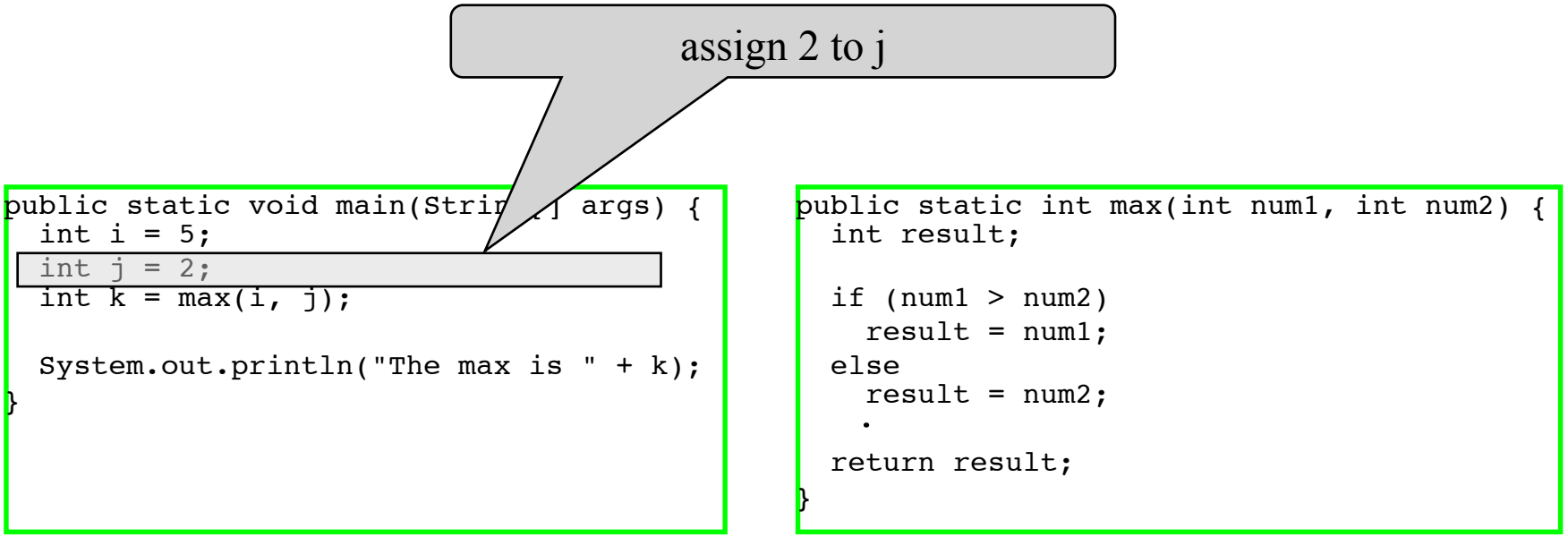
animation

Trace Method Invocation



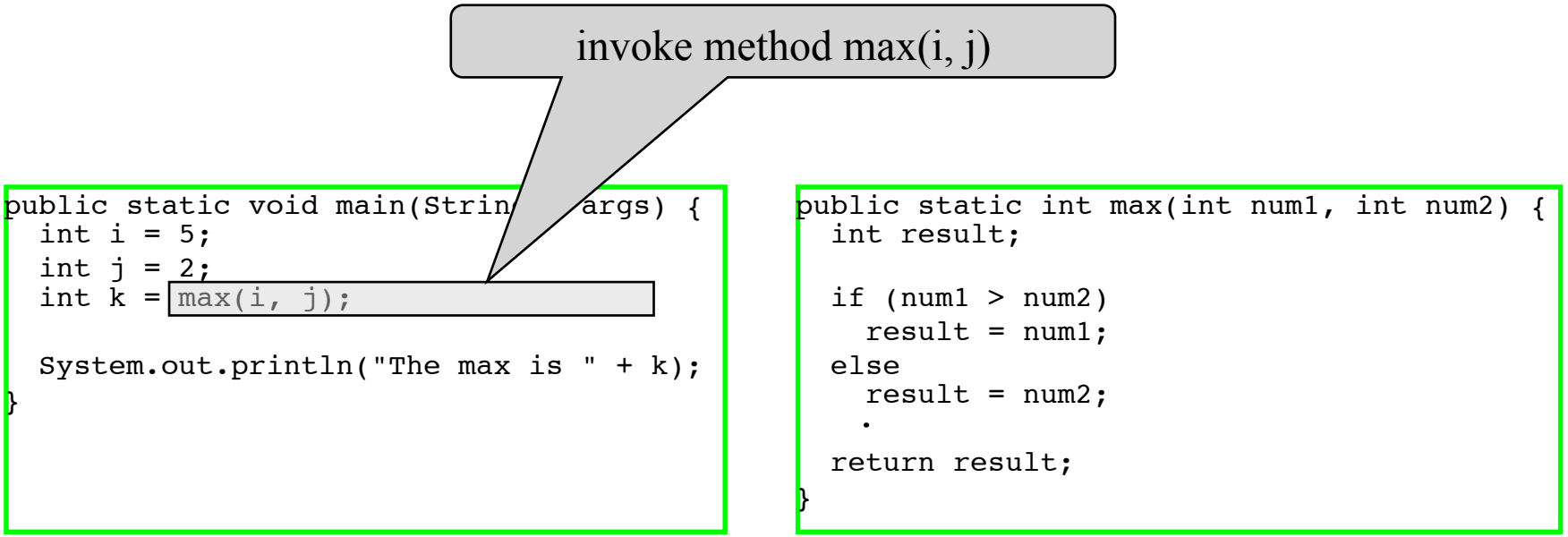
animation

Trace Method Invocation



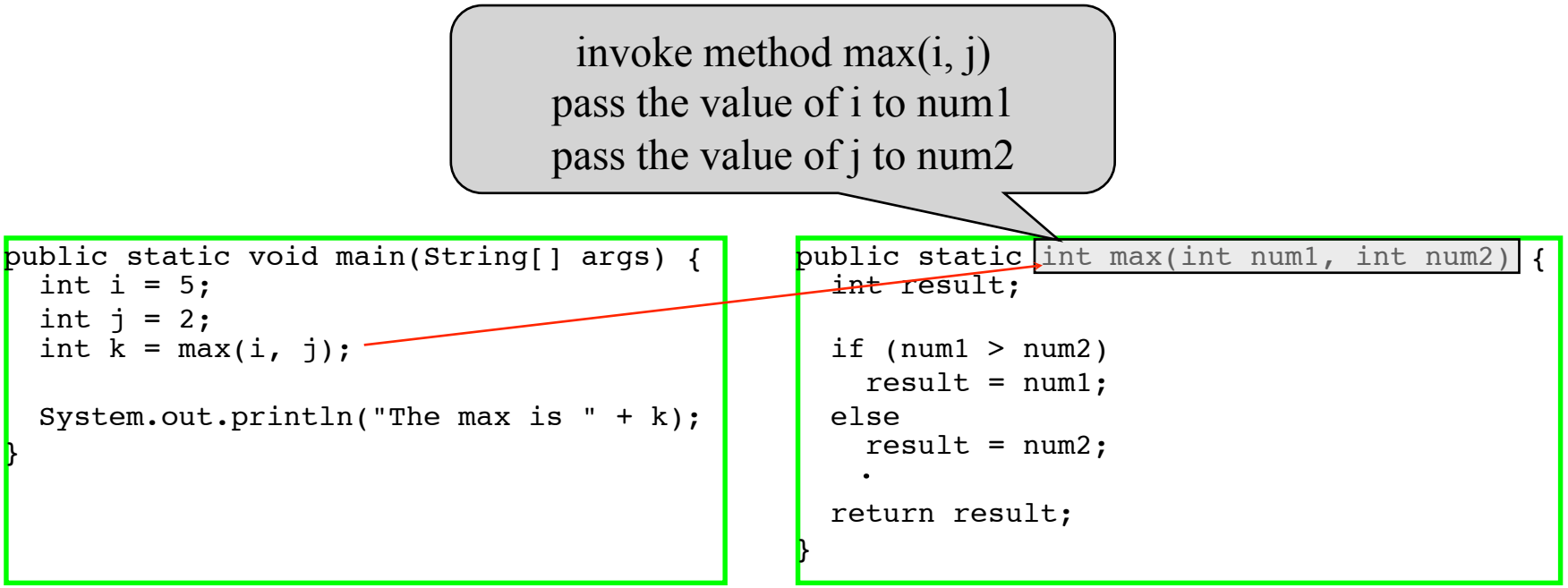
animation

Trace Method Invocation



animation

Trace Method Invocation



animation

Trace Method Invocation

declare variable result

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println("The max is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

animation

Trace Method Invocation

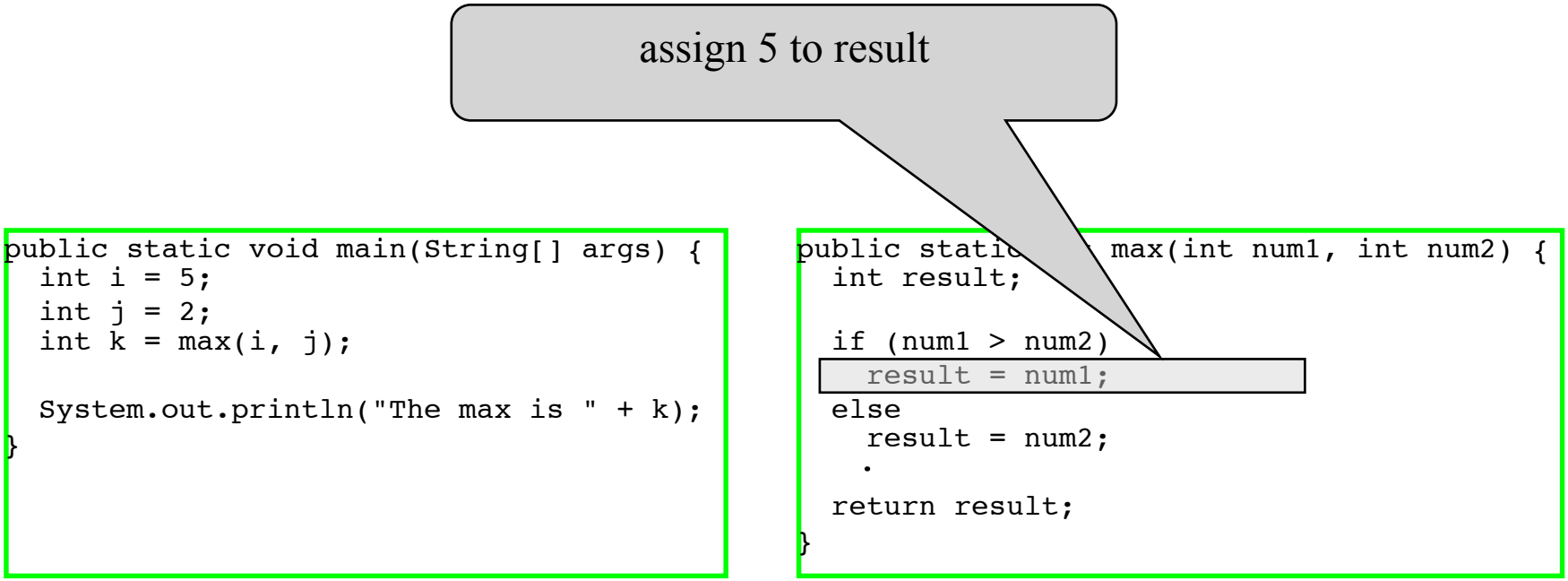
(num1 > num2) is true since
num1 is 5 and num2 is 2

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println("The max is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

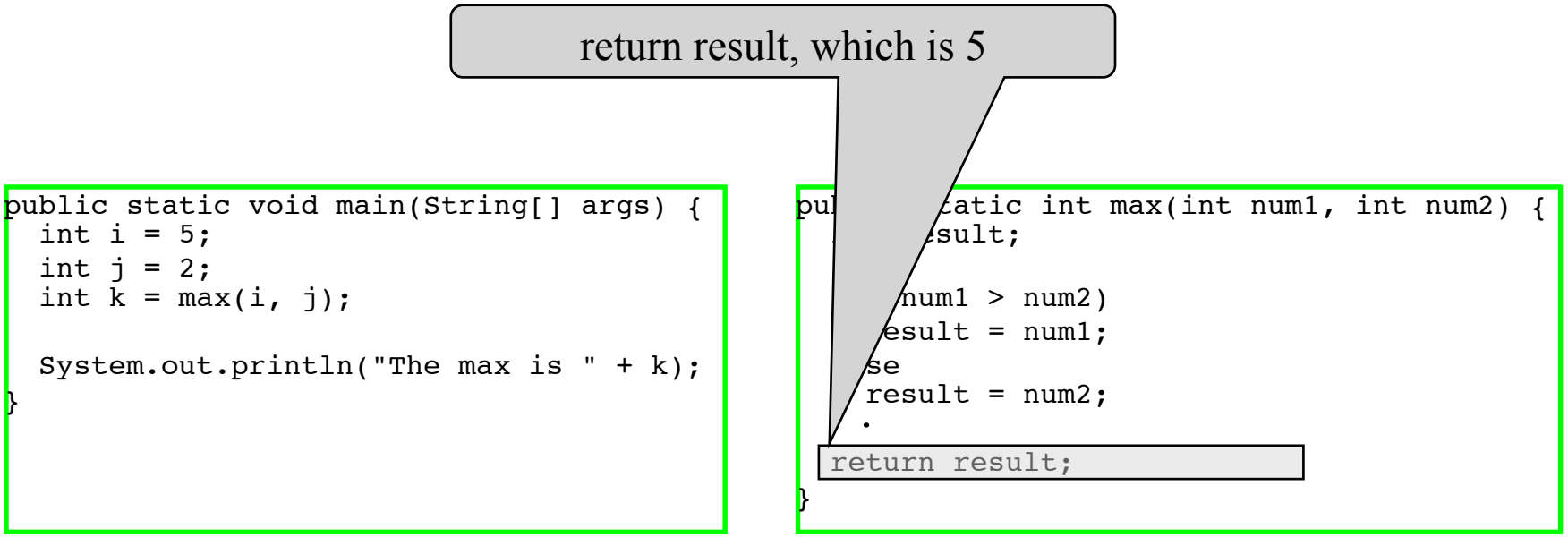
animation

Trace Method Invocation



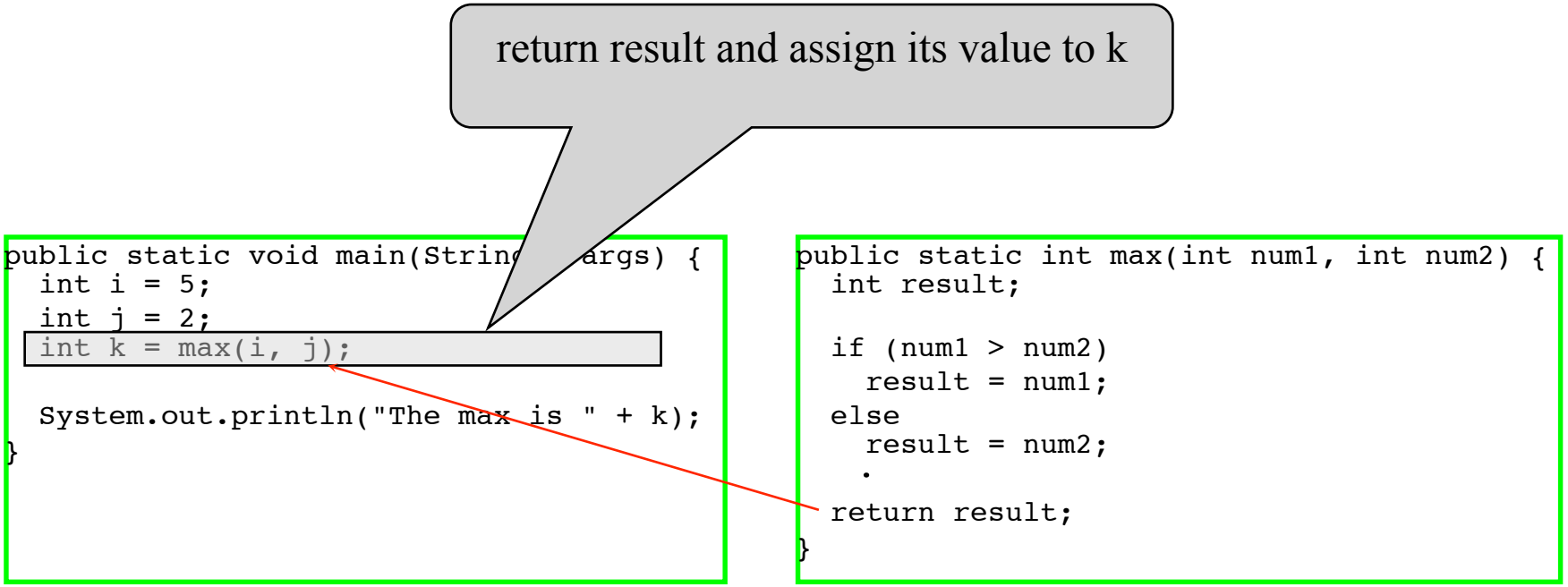
animation

Trace Method Invocation



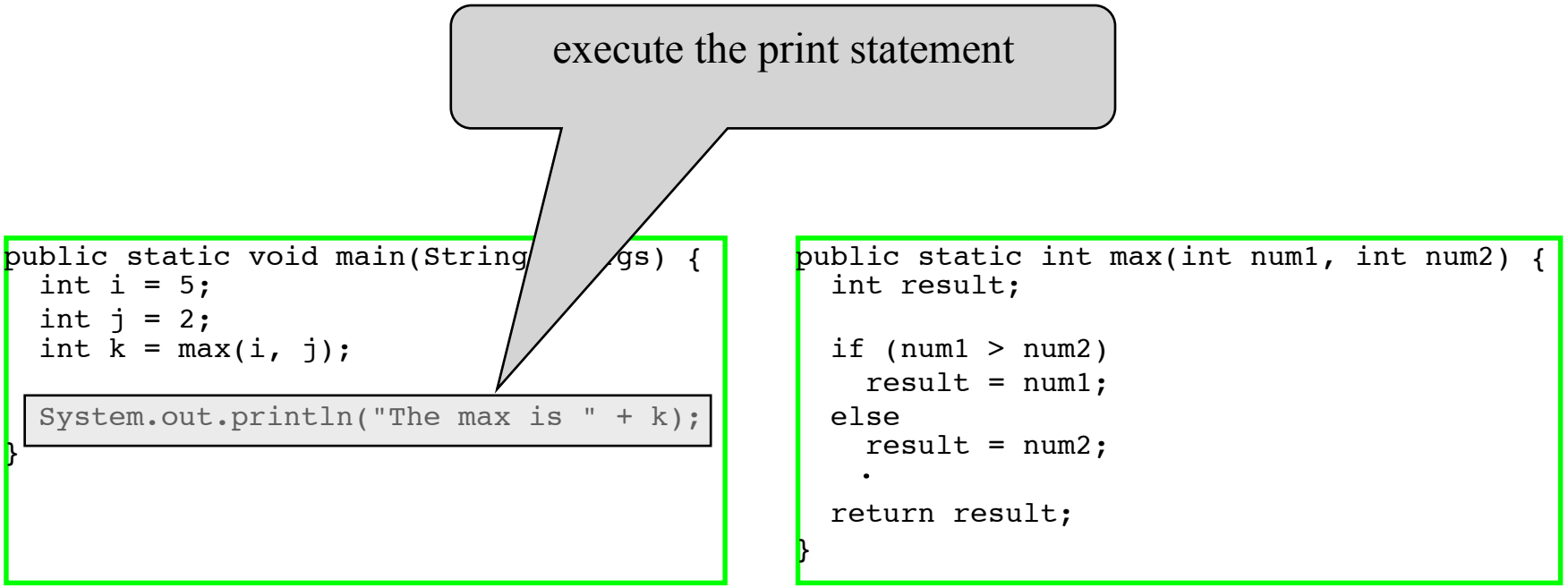
animation

Trace Method Invocation



animation

Trace Method Invocation



Defining Methods

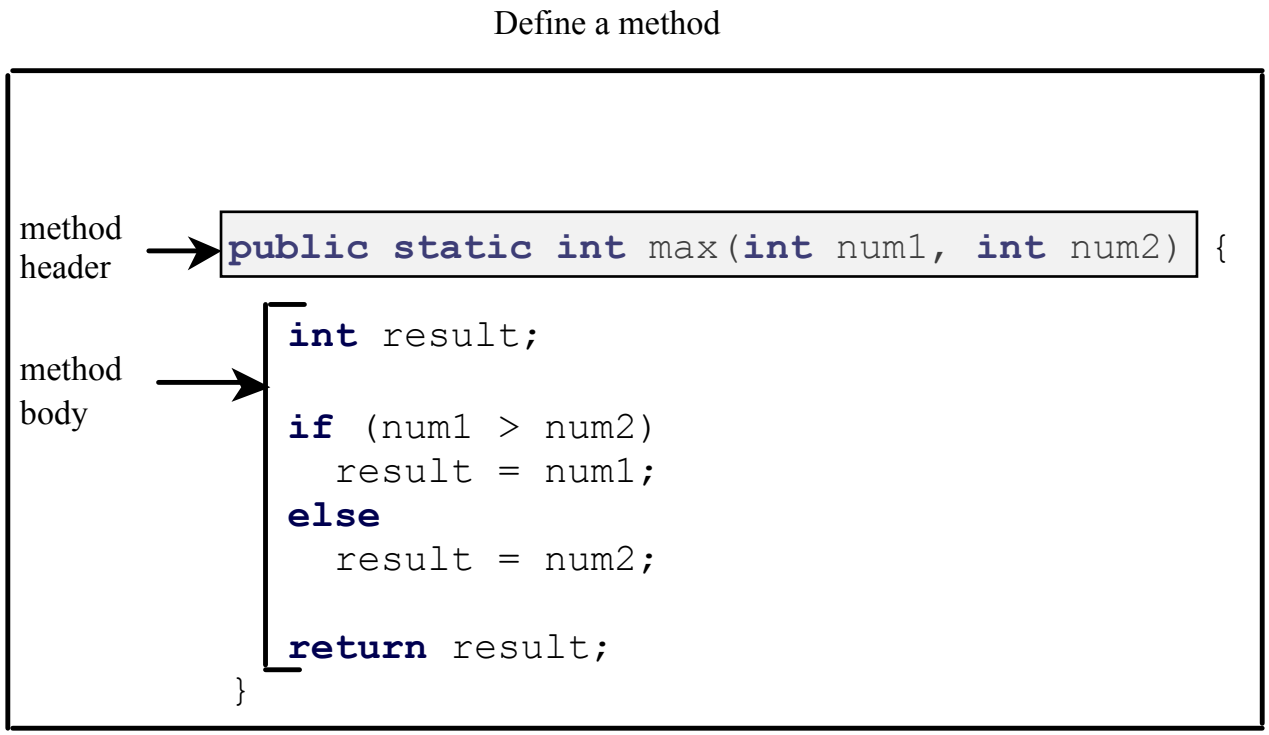
A method is a collection of statements that are grouped together to perform an operation.

Define a method

```
public static int max(int num1, int num2) {

  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

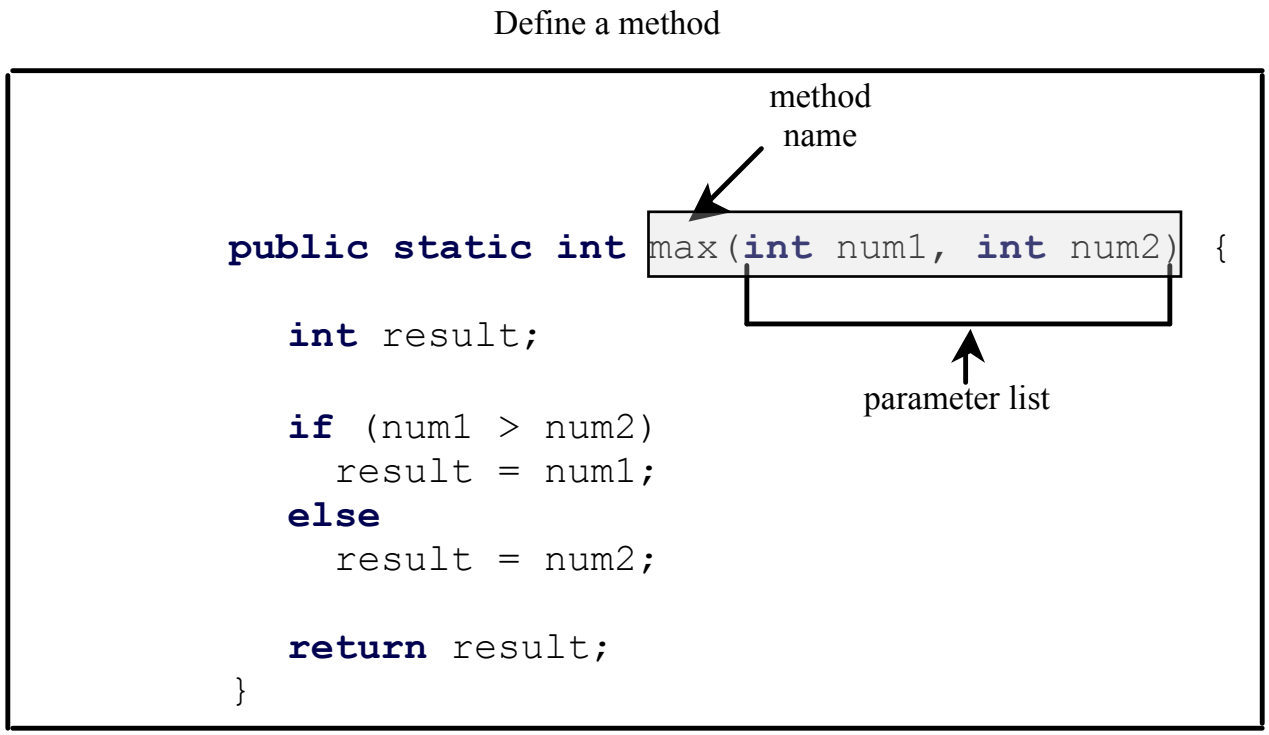
Defining Methods

A method definition consists of a *method header* and a *method body*.



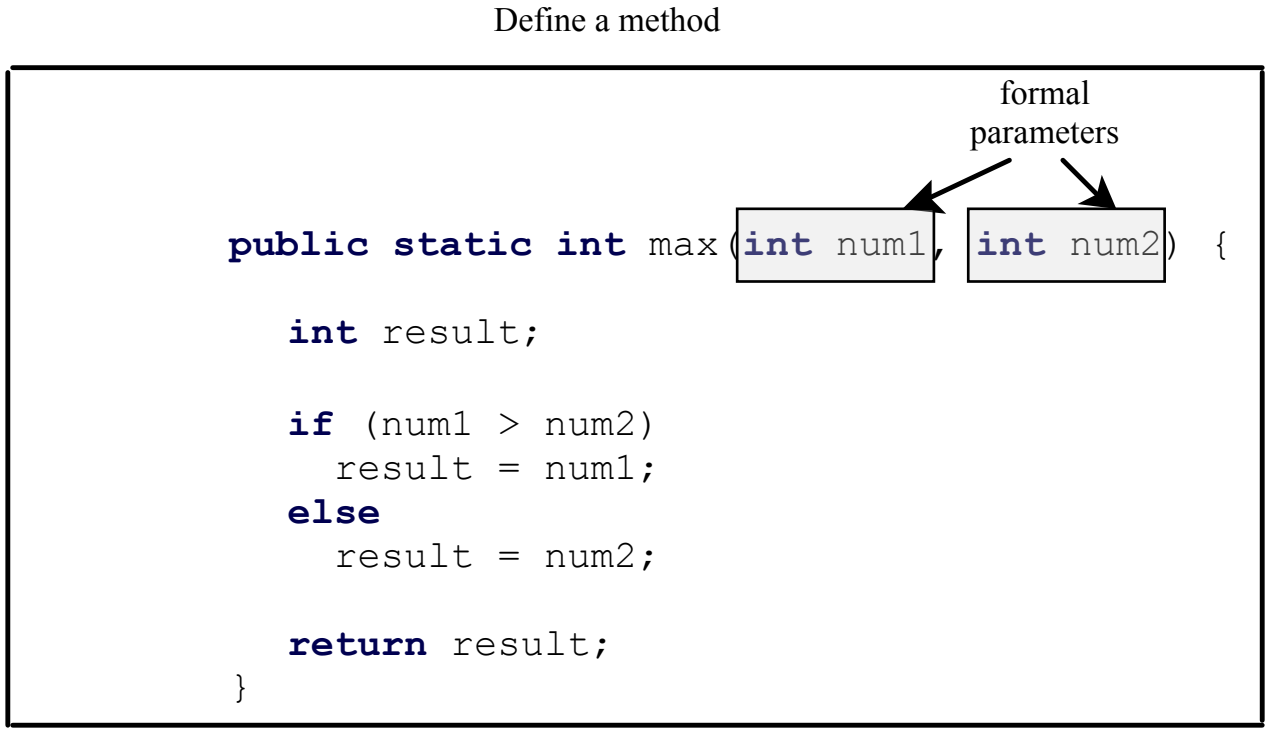
Method Signature

The *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*.

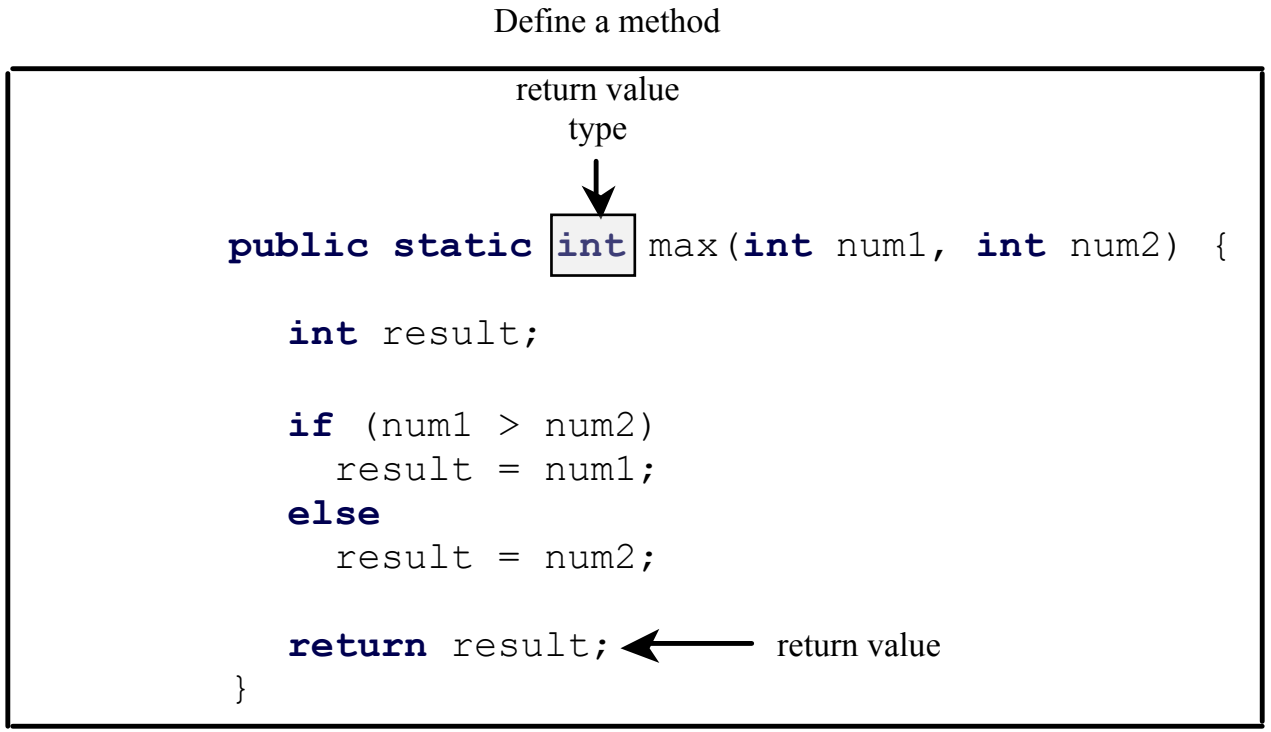


Return Value Type

A method may return a value. The *return value type* is the data type of the value the method returns.

If a method does not return a value, the return value type is the keyword `void`. For example, the return value type in the `main` method is `void`:

```
public static void main(String[] args)
```



Actual Parameters

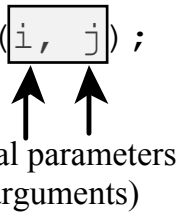
When a method is invoked, value(s) are passed to the parameter(s). These values are referred to as *actual parameters* or *arguments*.

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 i = 5;
int j = 2;
int k = max(i, j);
```



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?

What happens when you invoke the following method?

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

Passing Arguments by Value

The arguments are passed by value to parameters when invoking a method.

This is referred to as *pass-by-value*.

The value of the variable which is passed to the parameter is not affected, regardless of the changes made to the parameter inside the method.

Passing Arguments by Value

Example – passing arguments by value to methods…

TestPassByValue

Overloading Methods

Overloading methods enables you to define methods with the same name as long as their *signatures* are different:

- Remember – the method *name* and *parameter list* together constitute the *method signature*:

```
public static int max(int num1, int num2)
```

Methods that perform the same function with different types of parameters should be given the same name. Makes programs clearer and more readable.

The Java compiler determines which method to use based on the method signature.

Overloading Methods

Example – overloading the `max` method:

```
public static int max(int num1, int num2) {
  return (num1 > num2) ? num1 : num2;
}
```

```
public static double max(double num1, double num2) {
  return (num1 > num2) ? num1 : num2;
}
```

Suppose `max(3,4)` is invoked. Both methods are possible matches:

- The Java compiler finds the method that *best matches* a method invocation
- Since the method `max(int,int)` is a better match for `max(3,4)` than `max(double,double)`, it is used to invoke `max(3,4)`

Ambiguous Invocation

Sometimes there may be two or more possible matches for the 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

Example of 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) {
    return (num1 > num2) ? num1 : num2;
  }

  public static double max(double num1, int num2) {
    return (num1 > num2) ? num1 : num2;
  }
}
```

Ambiguous Invocation

Solution:

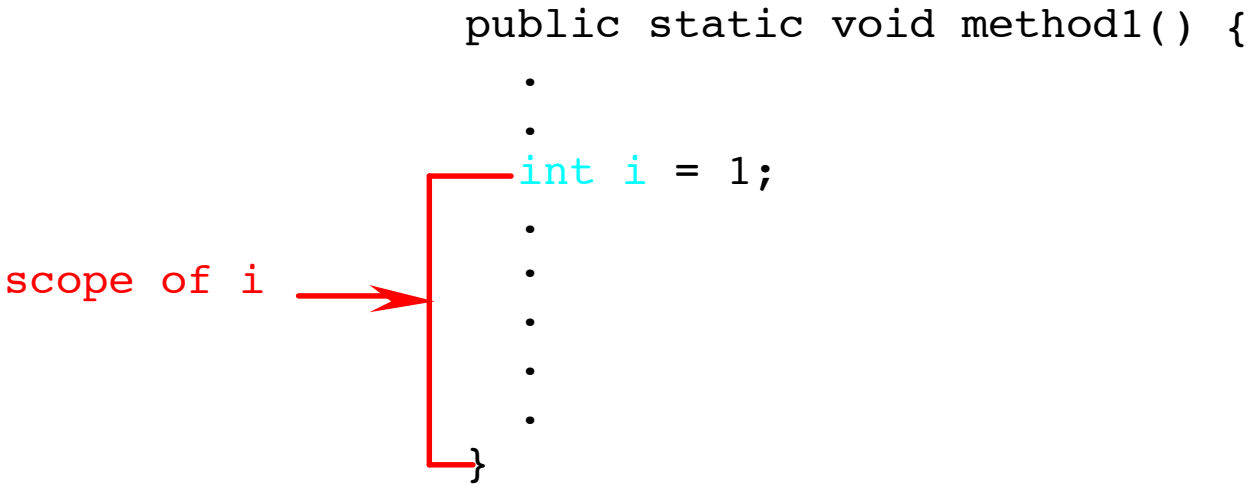
```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

  public static double max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
  }

  public static double max(double num1, double num2) {
    return (num1 > num2) ? num1 : num2;
  }
}
```

Scope of Local Variables

Local variable: a variable defined inside a method.

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

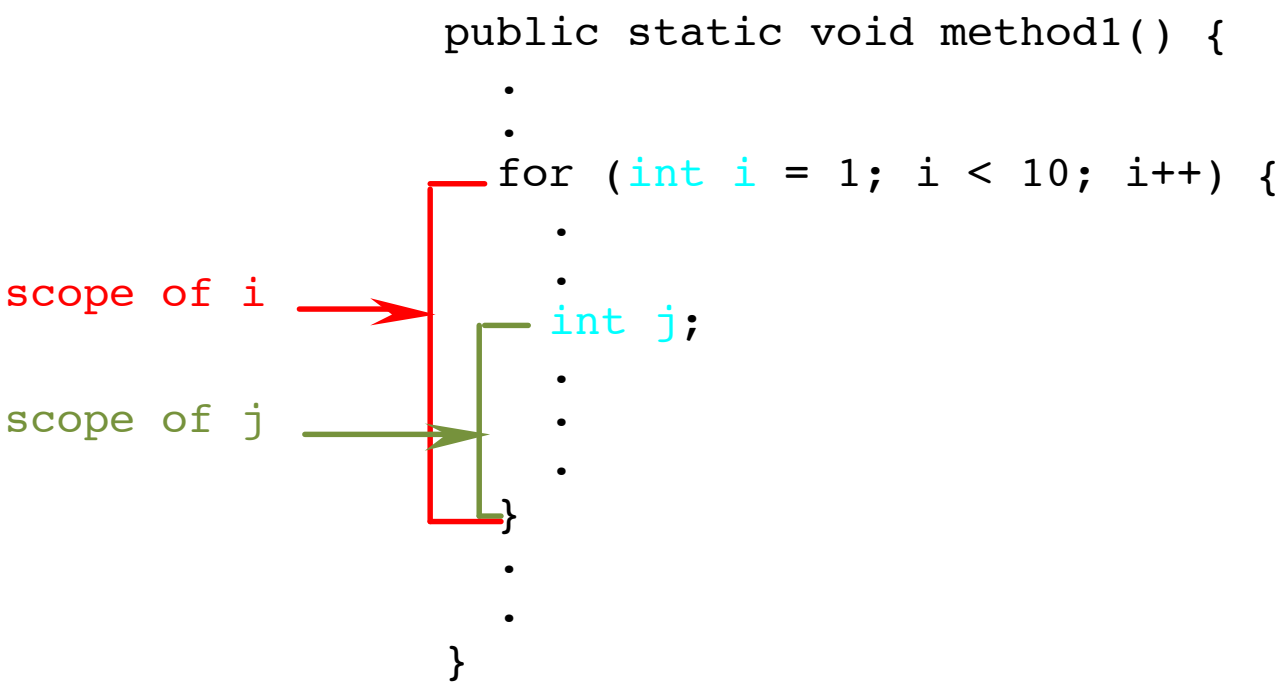
The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.



Scope of Local Variables

A variable declared in the initial action part of a `for` loop header has its scope in the entire loop.

A variable declared inside a `for` loop body has its scope limited to the loop body from its declaration and to the end of the `for` loop.



Scope of Local Variables

Variable `i` **can** be declared 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;
  }
}
```

Variable `i` **cannot** be declared in two **nesting** blocks:

```
public static void method2() {
  int i = 1;
  int sum = 0;

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

Modularizing Code

Methods can be used to reduce redundant coding and enable code reuse.

Methods facilitate the modularization of code, thereby improving the quality of the program.

Separates the functionality of a program into independent modules, such that each module contains everything necessary to execute only one aspect of the desired functionality.

Modularizing Code

By writing a method to obtain e.g. the maximum value of two integers, a number of advantages accrue:

- It isolates the problem for computing the maximum value from the rest of the code in the `main` method. Thus, the logic becomes clear and the program is easier to read.
- Any errors in computing the maximum value are confined to the `max` method, which narrows the scope of debugging.
- The `max` method can also be reused by other programs.

Method Abstraction

The key to developing software is to apply the concept of abstraction.

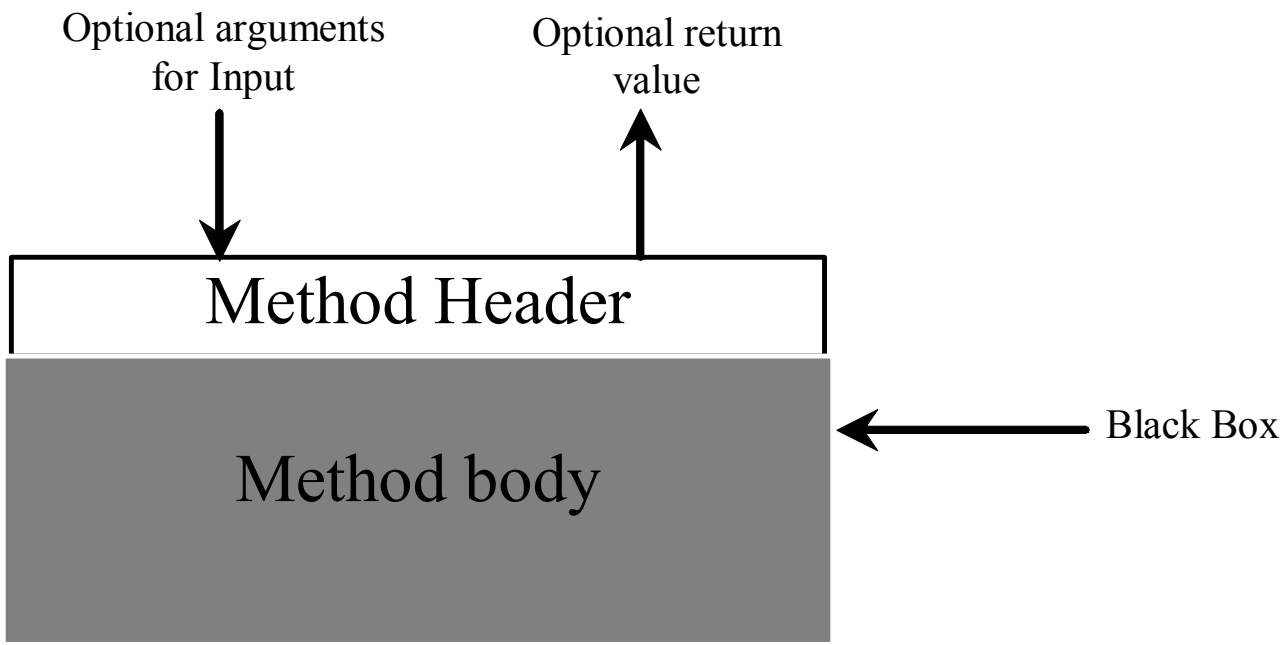
Method abstraction is achieved by separating the *use* of a method from its *implementation*.

The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client which invokes the method. This is also known as *information hiding* or *encapsulation*.

If you decide to change the implementation of the method, the client program will not be affected, provided that you do not change the method signature.

Method Abstraction

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 method implementation from the user.

Reduce complexity.

Stepwise Refinement

The concept of method abstraction can be applied to the process of developing programs.

When writing a large program to solve a complex problem:

- Use the *divide and conquer* strategy, also known as *stepwise refinement*, to decompose the problem into sub-problems...
- Sub-problems can be further decomposed into smaller, more manageable problems…
- A *design diagram* (aka *structure chart*) can be used to visualize this process.

The idea is not to focus on *implementation details*, rather to decompose and identify the successive steps involved in writing the program.

Case Study: Validating Credit Card Numbers

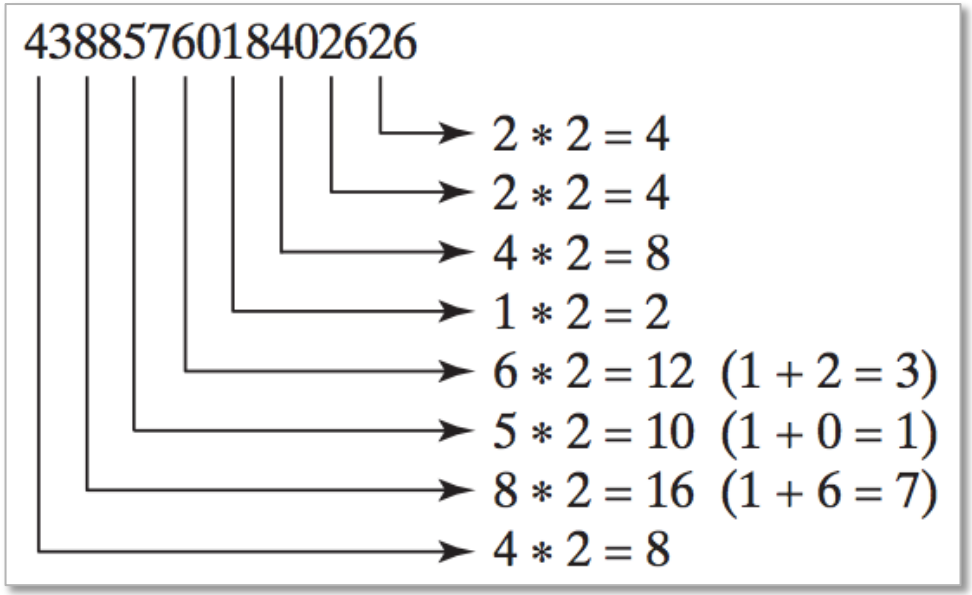
A case study to demonstrate stepwise refinement. Consider a program to validate credit card numbers...

Valid credit card numbers must satisfy certain criteria:

1) The number must have between 13 and 16 digits.
2) The number must have a valid prefix – i.e. it must start with any of the following:
   - 4 (for Visa cards)
   - 5 (for Master cards)
   - 37 (for American Express cards)
   - 6 (for Discover cards)
3) The number must satisfy the *Luhn check* (aka the *Mod 10 check)*.

Validating Credit Card Numbers

The *Luhn check*:



(i) Double every second digit from right to left.

If doubling of a digit gives a two-digit number, add the two digits to get a single-digit number.

Add all single-digit numbers:

$$4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37$$

(ii) Add all digits in the odd places from right to left in the card number:

$$6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38$$

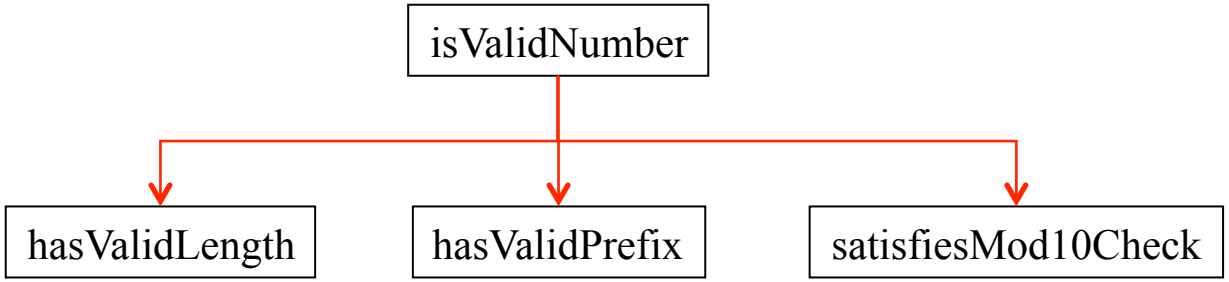
Sum the results from **(i)** and **(ii)** above:

$$37 + 38 = 75$$

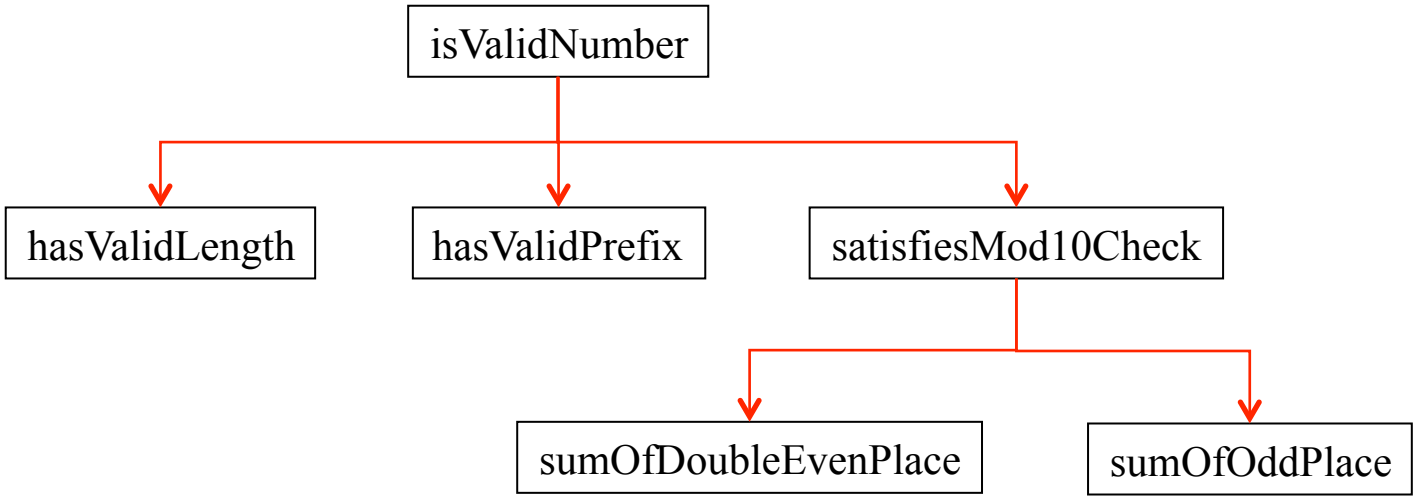
If the sum is divisible by 10, the card number is valid; otherwise, it is invalid.

Design Diagram

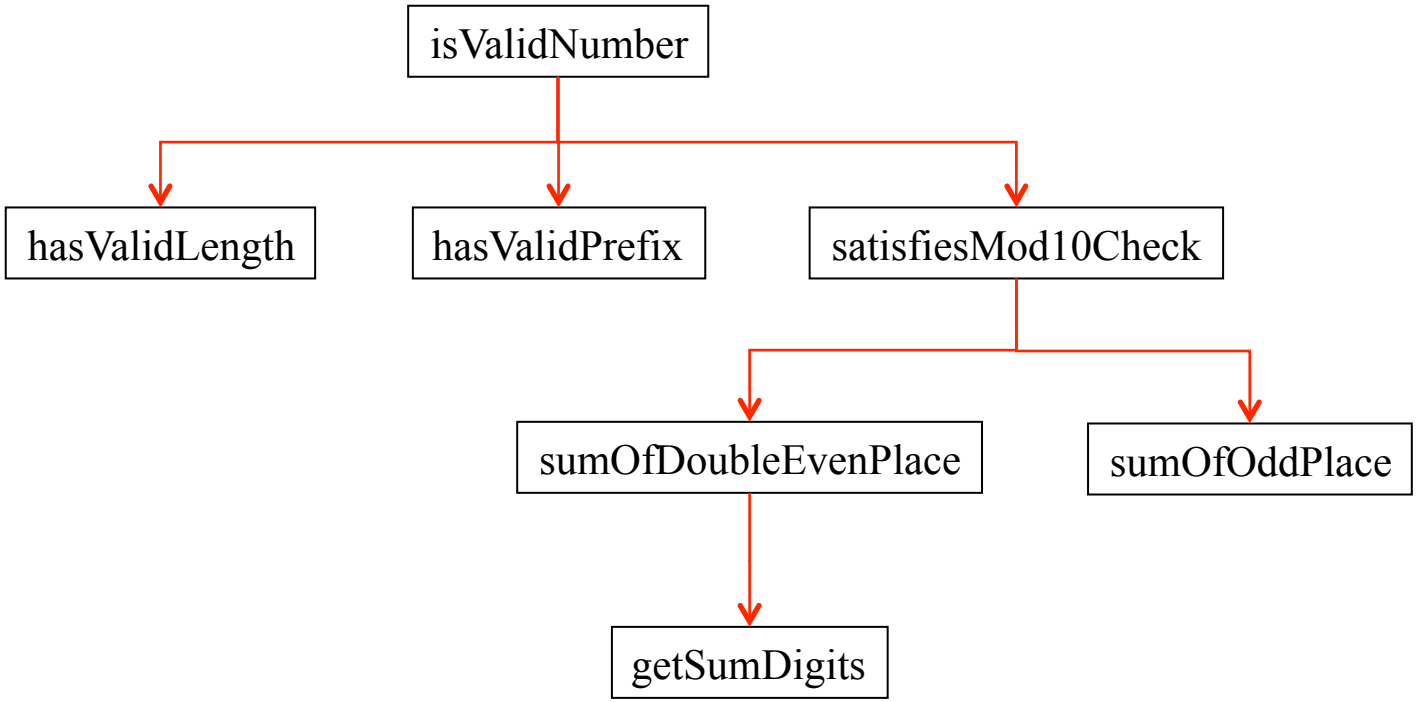
Design Diagram



Design Diagram



Design Diagram



Methods

```
// Return true if the card number is valid
public static boolean isValidNumber(String number)

// Return true if the card number has between 13 and 16 digits
public static boolean hasValidLength(String number)

// Return true if the card number has a valid prefix
public static boolean hasValidPrefix(String number)

// Return true if the Mod 10 check is satisfied
public static boolean satisfiesMod10Check(String number)

// Double every second digit from right to left and return sum
public static int sumOfDoubleEvenPlace(String number)

// Return sum of digits in odd places from right to left
public static int sumOfOddPlace(String number)

// Return this number if it is a single digit;
// otherwise return the sum of the two digits
public static int getSumDigits(int number)
```

Implementation: Top-Down

The top-down approach is to implement one method in the design diagram (structure chart) at a time from the top to the bottom.

Stubs can be used for the methods waiting to be implemented. A stub is a simple but incomplete version of a method.

The use of stubs enables you to test invoking the method from a caller.

Implement the top method first and then use a stub for the other methods…

For example, your program to validate credit card numbers might begin like this…

ValidateCCStubs

Implementation: Bottom-Up

The bottom-up approach is to implement one method in the design diagram (structure chart) at a time from the bottom to the top.

For each method implemented, write a test program to test it.

Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy.

Benefits of Stepwise Refinement

Simpler programs

Reusing methods

Easier developing, debugging, and testing

Better facilitates teamwork

Case Study: Implementing a version of the `Math` class

The Java API provides many useful classes to, for example, perform mathematical operations (`Math` class) and manipulate characters (`Character` class).

In this case study, we will implement a new version of the `Math` class.

The objective is to move beyond writing the kind of programs we have considered until now – i.e. programs with a `main` method (and possibly other methods) that simply execute a set of statements.

The new version of the `Math` class we will write could, for example, be made available to assist other programmers in their work.

MyMath TestMyMath

Next Topics

Chapters 7 & 8:

- Single-dimensional & multidimensional arrays…