

# 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

```
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);
```



# 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);
```



# Solution

```
public static int sum(int i1, int i2) {  
    int sum = 0;  
    for (int i = i1; i <= i2; i++)  
        sum += i;  
    return sum;  
}
```

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

# 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

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;  
}
```

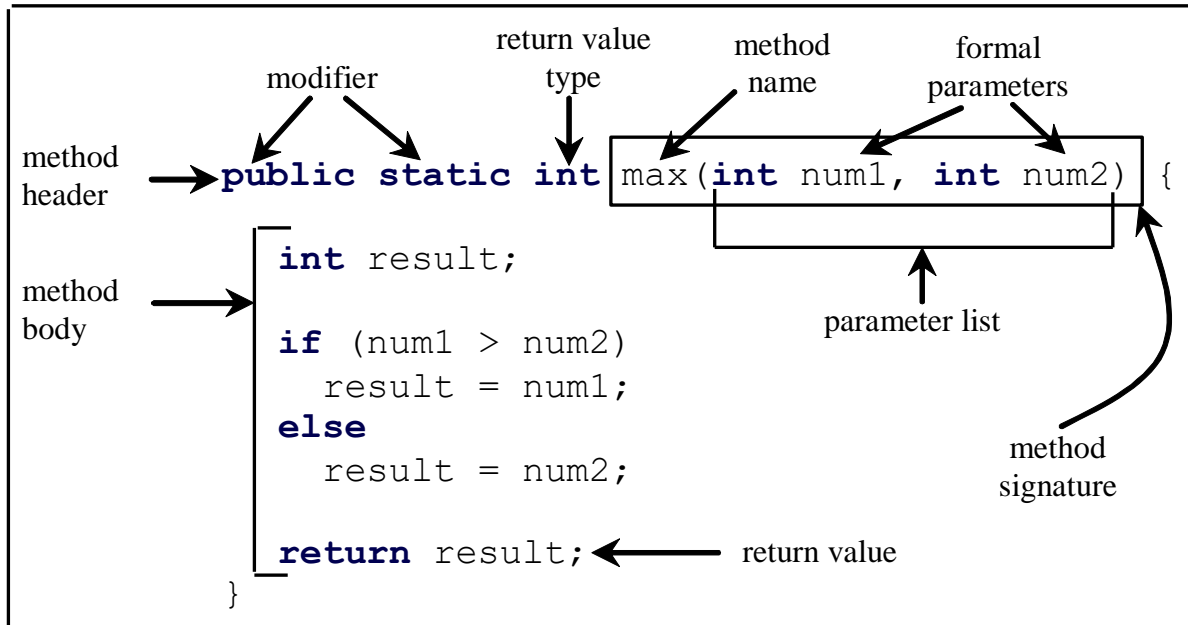
Invoke a method

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

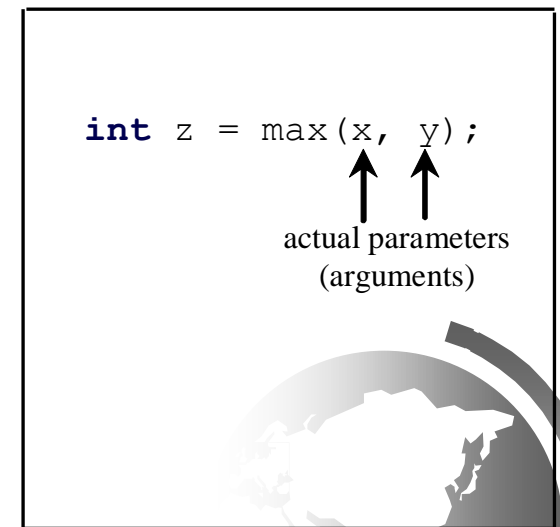
# Defining Methods

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

Define a method



Invoke a method



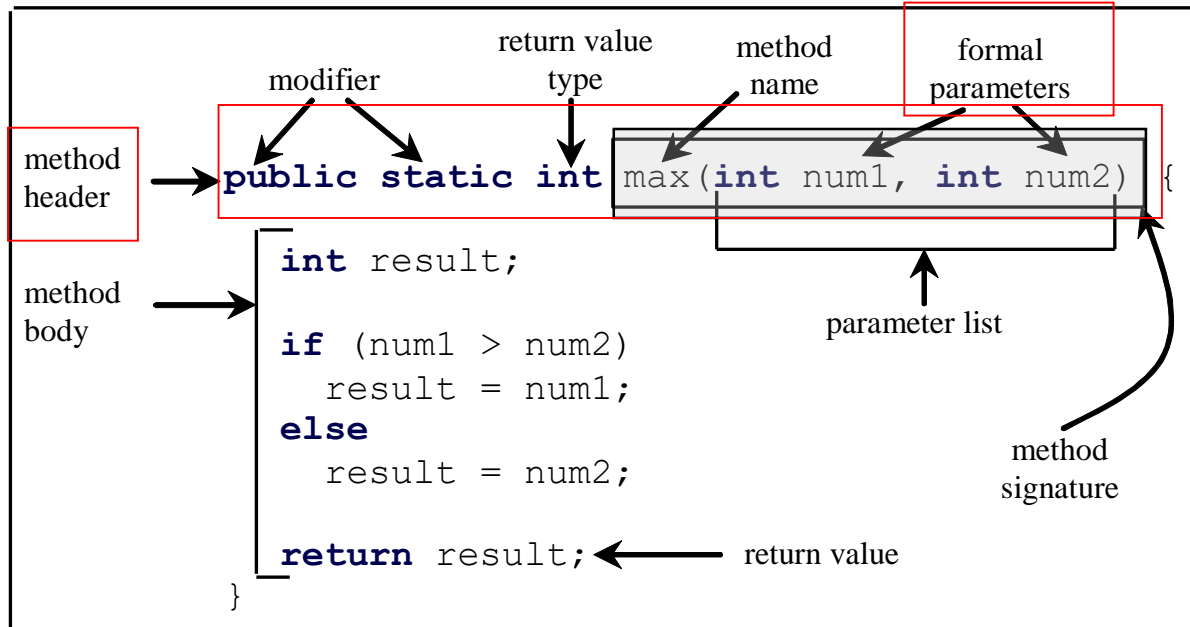




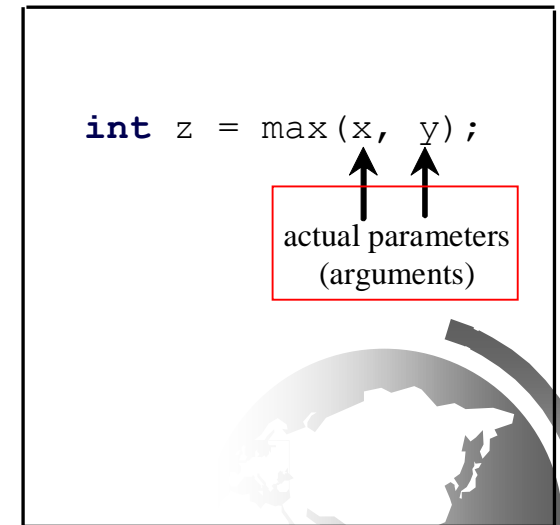
# Method Signature

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

Define a method



Invoke a method

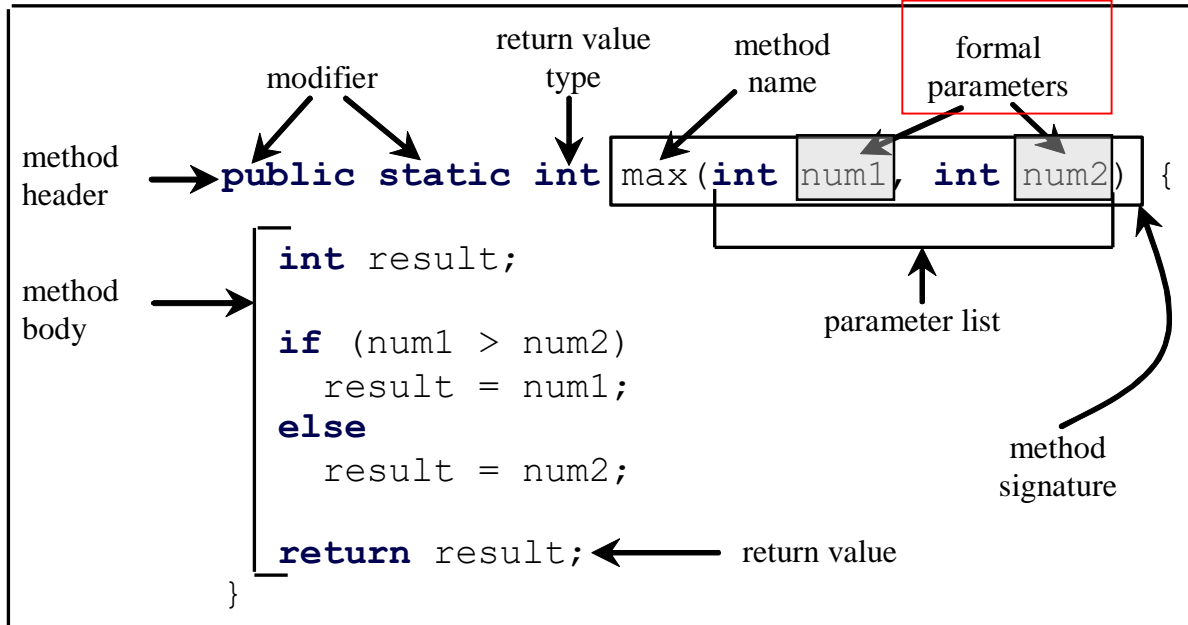




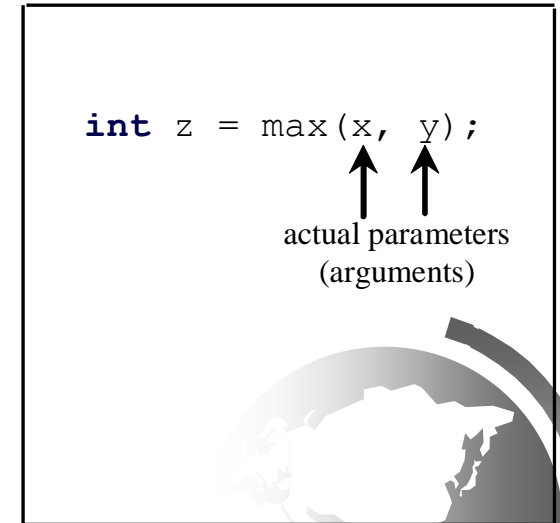
# Formal Parameters

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

Define a method



Invoke a method

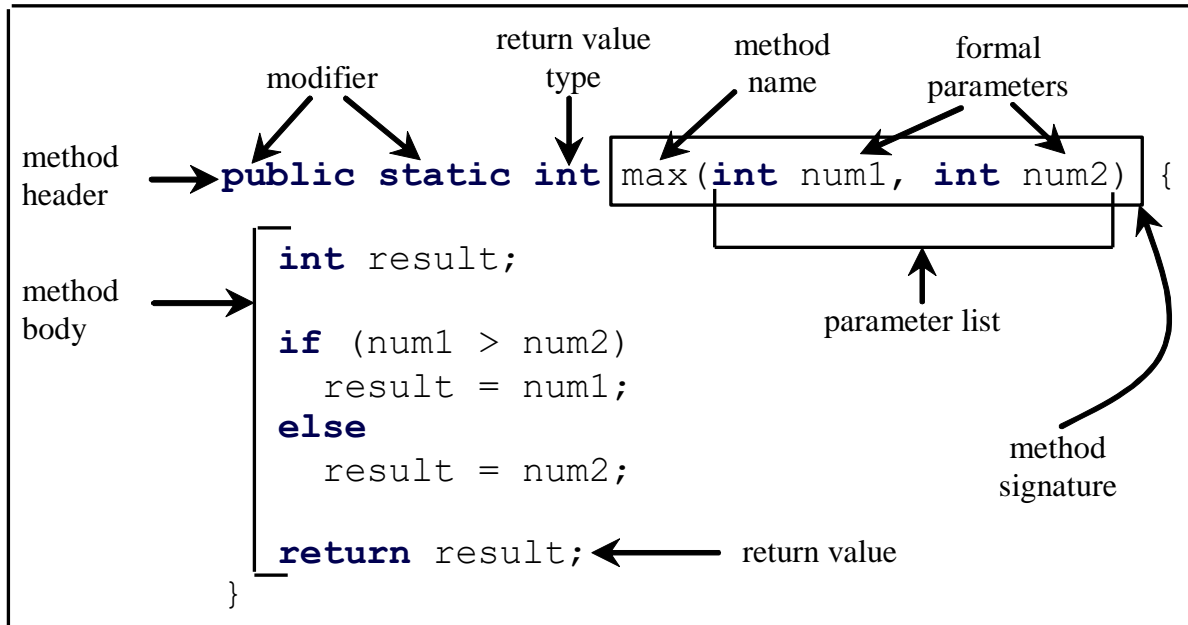




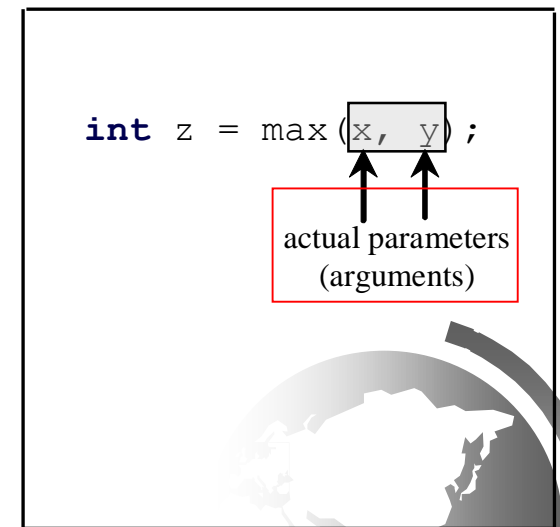
# Actual Parameters

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

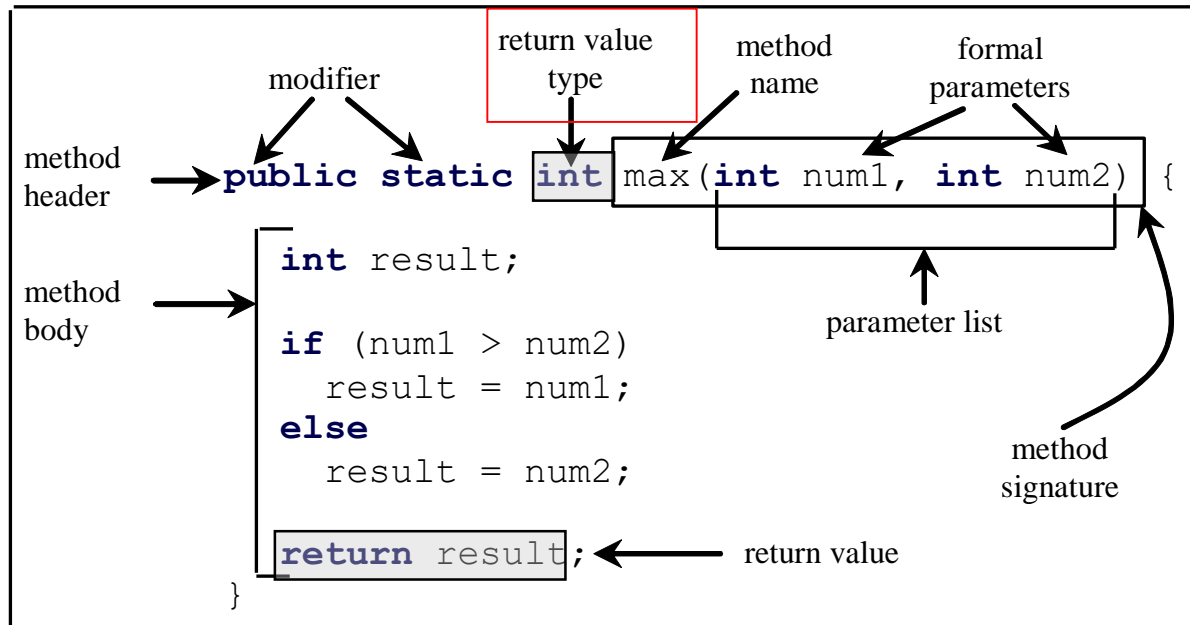




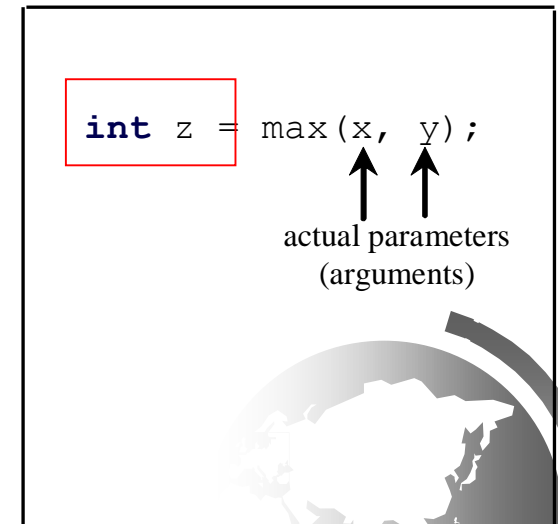
# Return Value Type

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

Testing the `max` method

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

Animation

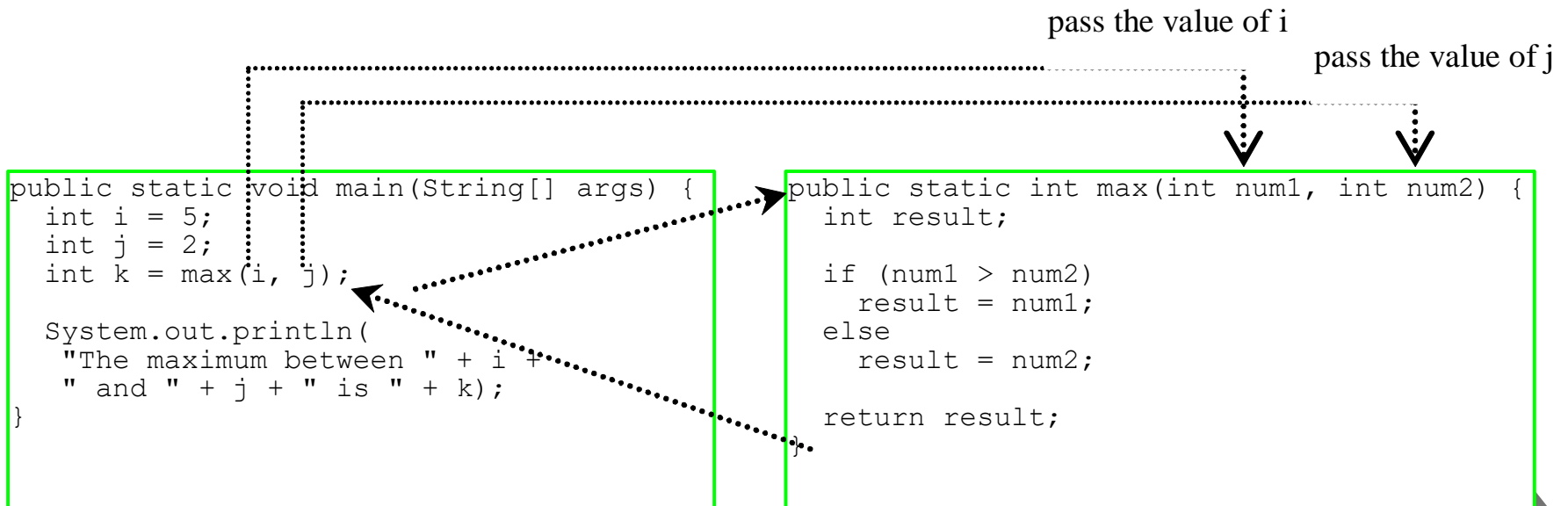


TestMax

Run



# Calling Methods, cont.



# Trace Method Invocation

i is now 5

```
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;  
}
```



# Trace Method Invocation

j is now 2

```
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;  
}
```





# Trace Method Invocation

invoke max(i, j)

```
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;  
}
```



# Trace Method Invocation

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

```
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;  
}
```



# 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 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;  
}
```



# 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 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;  
}
```



# Trace Method Invocation

result is now 5

```
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;  
}
```



# Trace Method Invocation

return result, which is 5

```
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;  
}
```



# Trace Method Invocation

return max(i, j) and assign the  
return value to 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;  
}
```



# Trace Method Invocation

Execute the print statement

```
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;  
}
```







# 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 (**no "else" part**).

```
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)

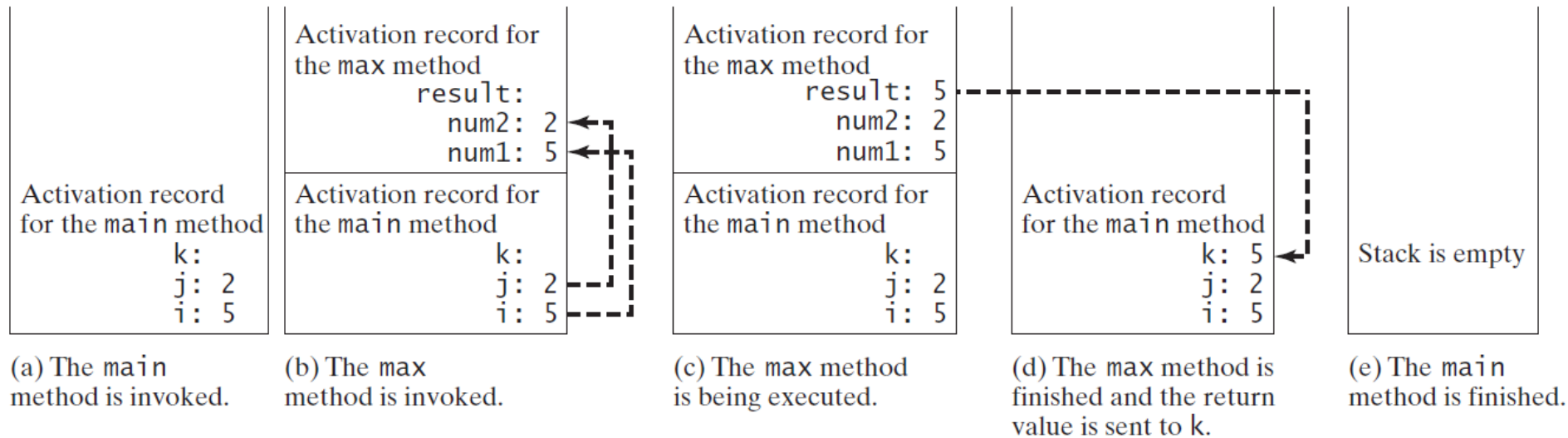
To fix this problem, delete if ( $n < 0$ ) in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

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



# Call Stacks





# 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

The main method  
is invoked.



# 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

The main method  
is invoked.



# 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;  
}
```

Space required for the  
main method

k:  
j: 2  
i: 5

The main method  
is invoked.



# Trace Call Stack

Invoke max(i, j)

```
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;  
}
```

Space required for the  
main method

k:  
j: 2  
i: 5

The main method  
is invoked.

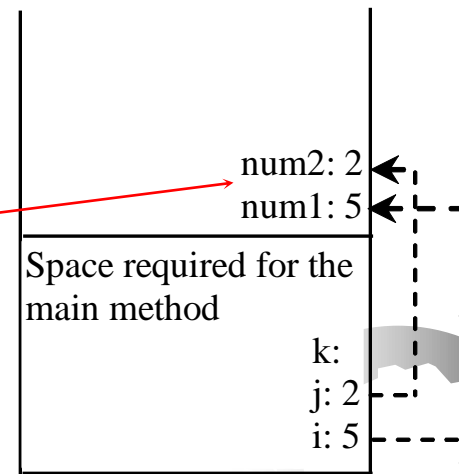


# Trace Call Stack

pass the values of i and j to num1  
and num2

```
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;  
}
```

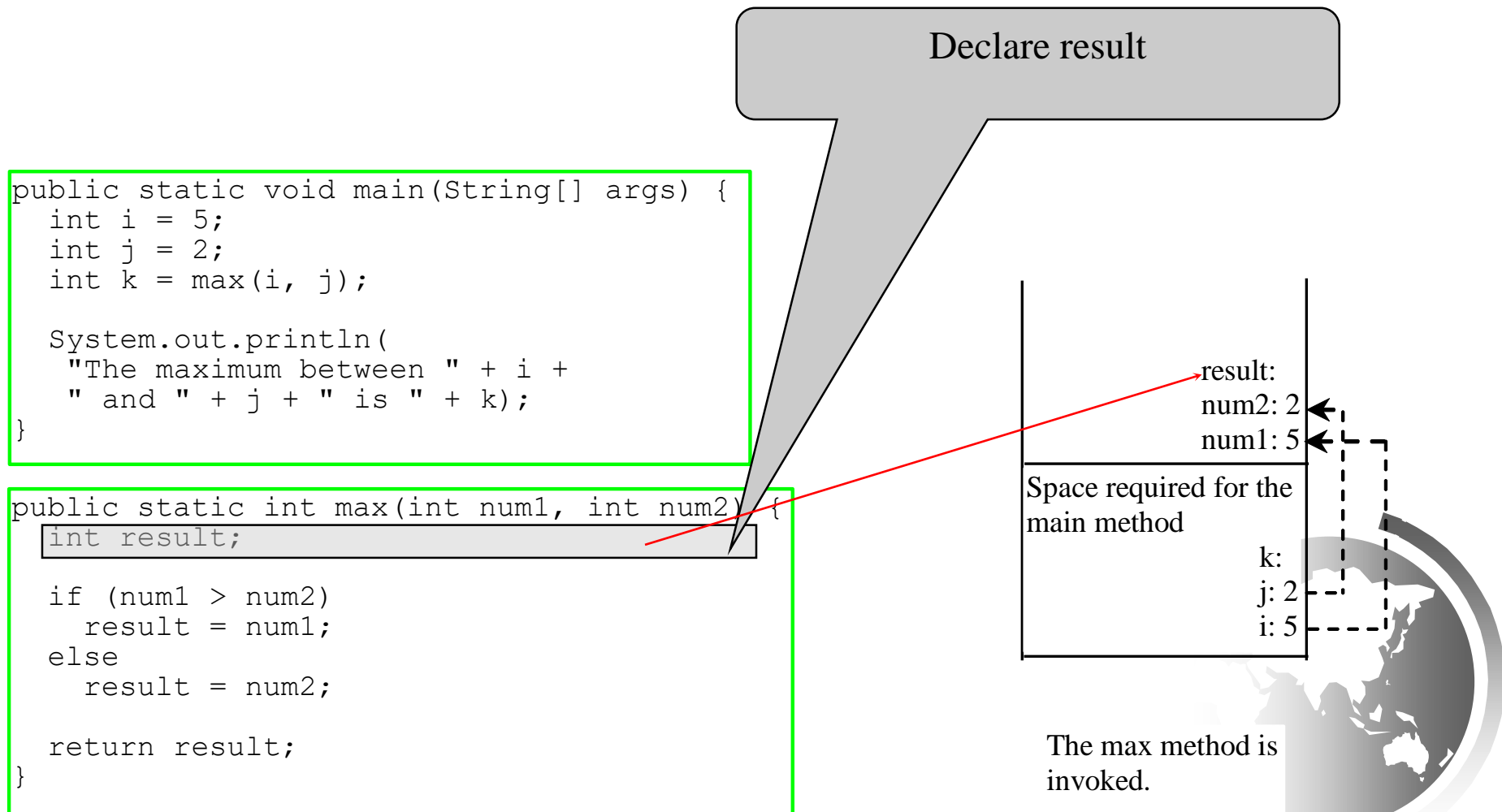


The max method is  
invoked.





# Trace Call Stack



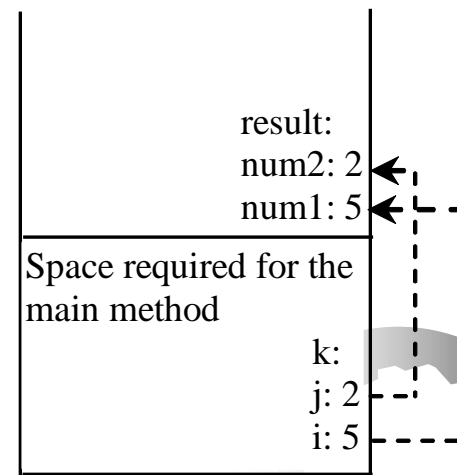


# Trace Call Stack

```
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);  
}
```

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

(num1 > num2) is true



The max method is  
invoked.

# 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);
}
```

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

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

    return result;
}
```

```
Assign num1 to result
```

### Space required for the max method

```
result: 5
num2: 2
num1: 5
```

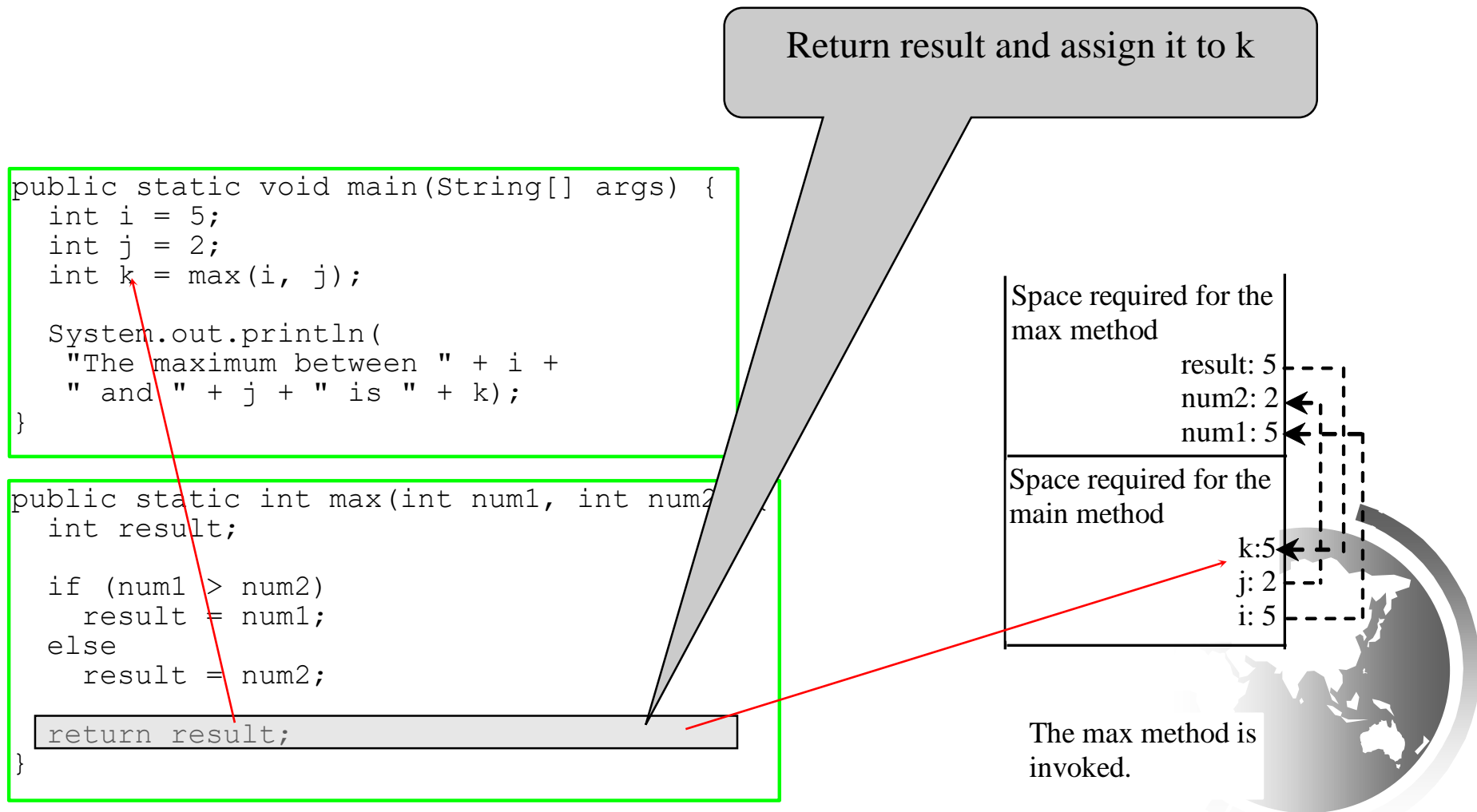
Space required for the  
main method

```
k:
j: 2
i: 5
```

The max method is invoked.



# Trace Call Stack





# Trace Call Stack

Execute print statement

```
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;  
}
```

Space required for the  
main method

k:5  
j: 2  
i: 5

The main method  
is invoked.



# void Method Example

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



TestVoidMethod

Run



TestReturnGradeMethod

Run

# 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

This program demonstrates passing values to the methods.



Increment

Run





# Pass by Value

Testing Pass by value

This program demonstrates passing values to the methods.

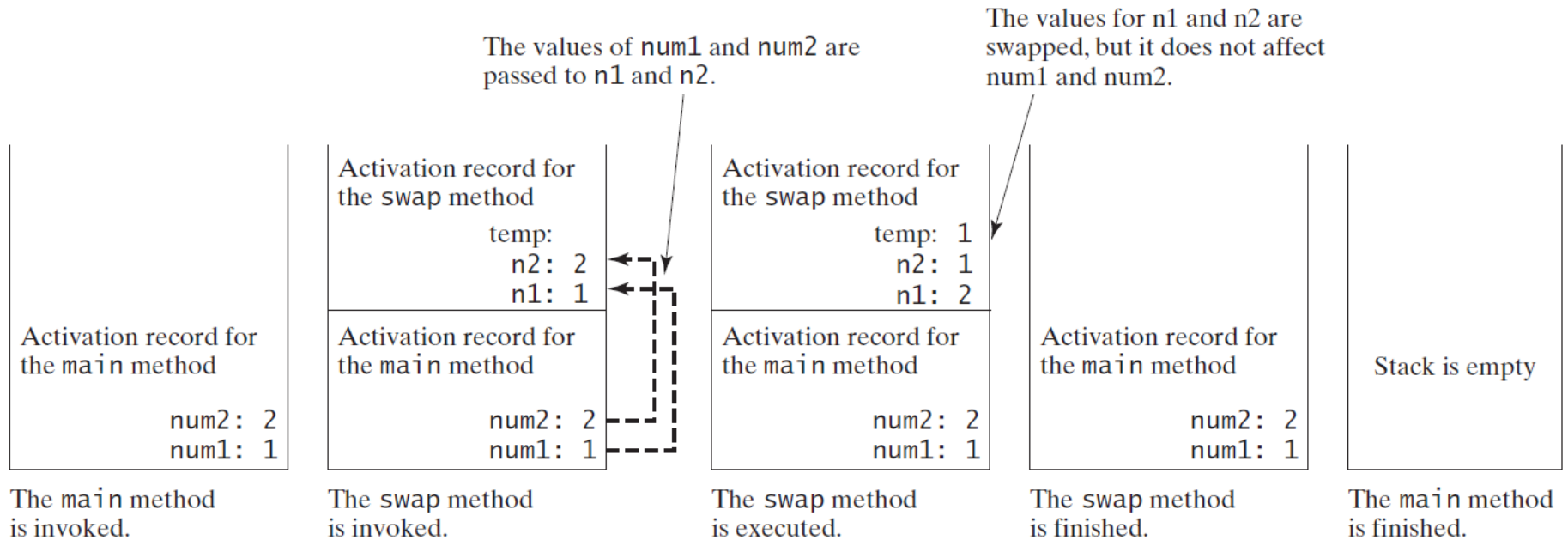


TestPassByValue

Run



# Pass by Value, cont.



# Modularizing Code

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.



GreatestCommonDivisorMethod

Run



PrimeNumberMethod

Run



# Case Study: Converting Hexadecimals to Decimals

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

ABCD =>

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

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

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



Hex2Dec

Run



# Overloading Methods

## Overloading the max Method

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



TestMethodOverloading

Run



```

public class TestMethodOverloading {
    /** Main method */
    public static void main(String[] args) {
        // Invoke the max method with int parameters
        System.out.println("The maximum of 3 and 4 is "
            + max(3, 4));

        // Invoke the max method with the double parameters
        System.out.println("The maximum of 3.0 and 5.4 is "
            + max(3.0, 5.4));

        // Invoke the max method with three double parameters
        System.out.println("The maximum of 3.0, 5.4, and 10.14 is "
            + max(3.0, 5.4, 10.14));
    }

    /** Return the max of two int values */
    public static int max(int num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }

    /** Find the max of two double values */
    public static double max(double num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }

    /** Return the max of three double values */
    public static double max(double num1, double num2, double num3) {
        return max(max(num1, num2), num3);
    }
}

```





# Ambiguous Invocation

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;  
    }  
}
```

**Only allow one signature to match**

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







# Scope of Local Variables

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

**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.



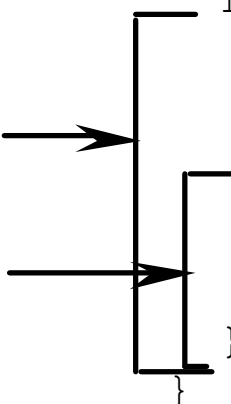
# ★ Scope of Local Variables, cont.

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;  
    }  
}
```

**The two “i” are not the same**



# ★ 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; // change: x = 0;  
        x += i;  
    }  
}
```

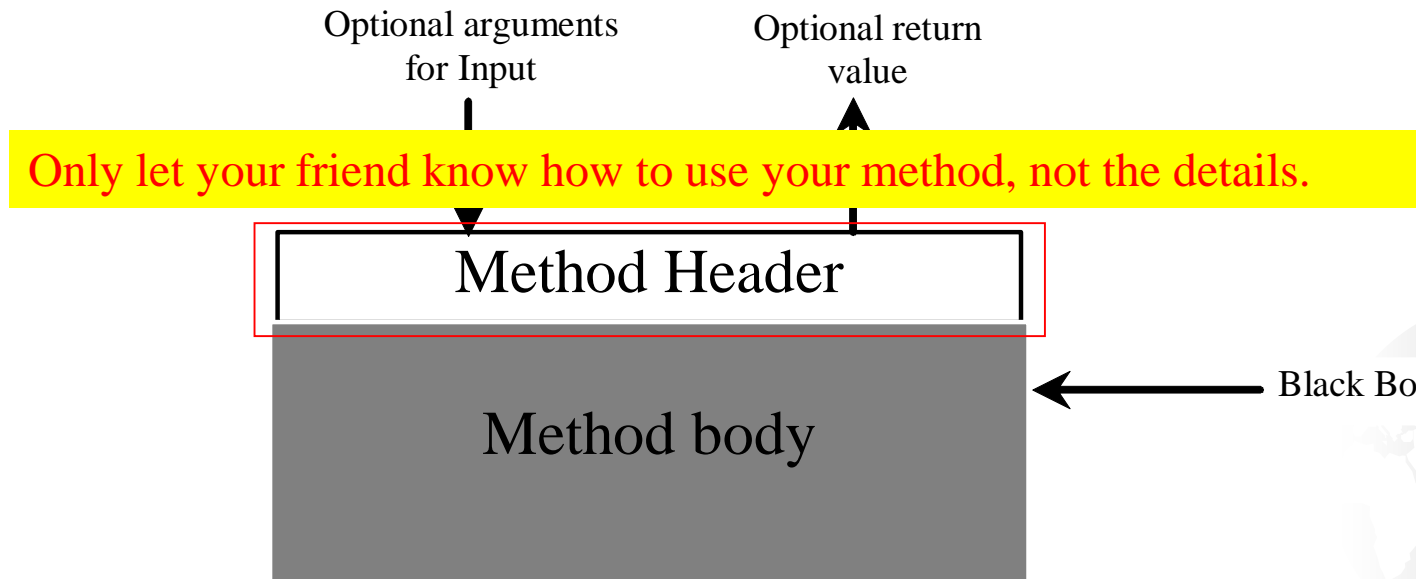
**The two “x” are not the same.  
The second “x” is a new variable.**





# 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**.



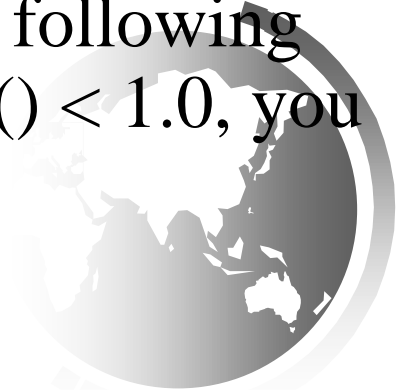


# Case Study: Generating Random Characters

Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them.

As introduced in Section 2.9, each character has a unique Unicode between 0 and FFFF in hexadecimal (65535 in decimal). To generate a random character is to generate a random integer between 0 and 65535 using the following expression: (note that since  $0 \leq \text{Math.random()} < 1.0$ , you have to add 1 to 65535.)

```
(int)(Math.random() * (65535 + 1))
```



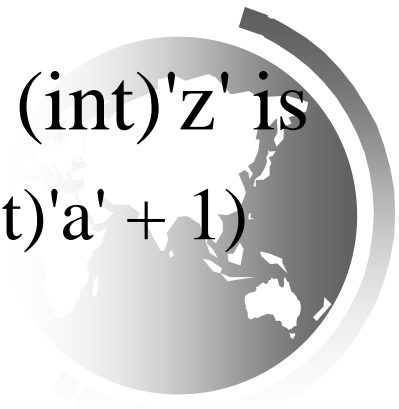
# Case Study: Generating Random Characters, cont.

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))`



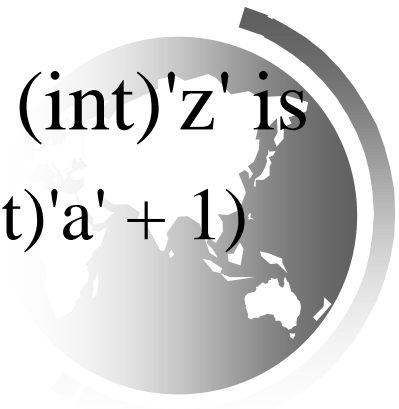
# Case Study: Generating Random Characters, cont.

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');  
}  
}
```



RandomCharacter



TestRandomCharacter



Run



# Stepwise Refinement (Optional)

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.





# PrintCalendar Case Study

Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.

```
Command Prompt
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
      April 2009
-----
Sun Mon Tue Wed Thu Fri Sat
      1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
 26 27 28 29 30

C:\book>
```



PrintCalendar

Run





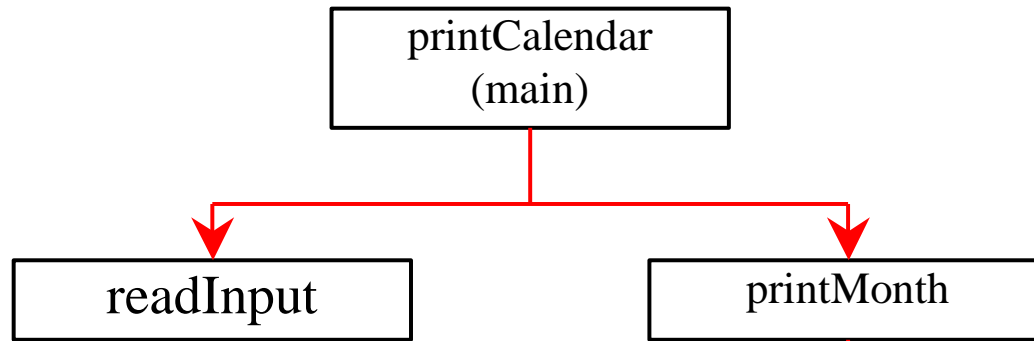


# Design Diagram

printCalendar  
(main)



# Design Diagram



Command Prompt

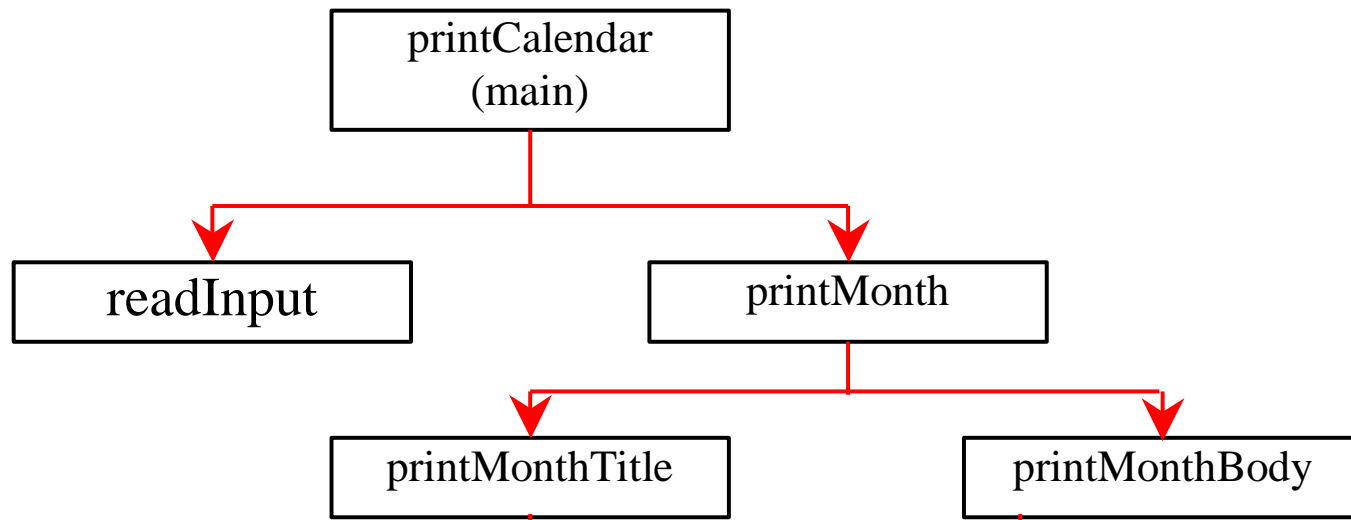
```
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
    April 2009
-----
Sun Mon Tue Wed Thu Fri Sat
      1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

C:\book>
```

The screenshot shows a Windows Command Prompt window. The user has entered the command `java PrintCalendar`. The program prompts for a full year (2009) and a month (4). It then displays the month name (April) and the year (2009). Below this, it shows a calendar grid for April 2009, with days of the week as headers and dates 1 through 30 arranged in rows. The prompt `C:\book>` is visible at the bottom.



# Design Diagram

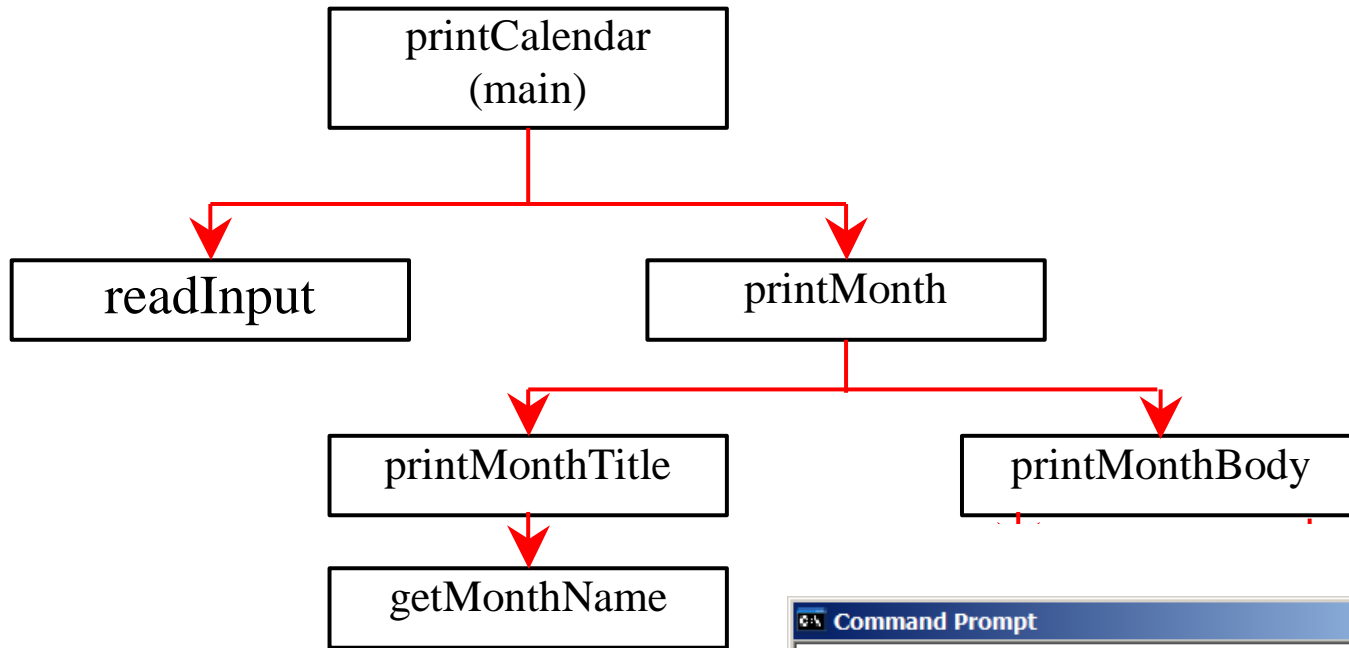


```
Command Prompt
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
      April 2009
-----
Sun Mon Tue Wed Thu Fri Sat
          1  2  3  4
 5   6   7   8   9  10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30

C:\book>
```



# Design Diagram

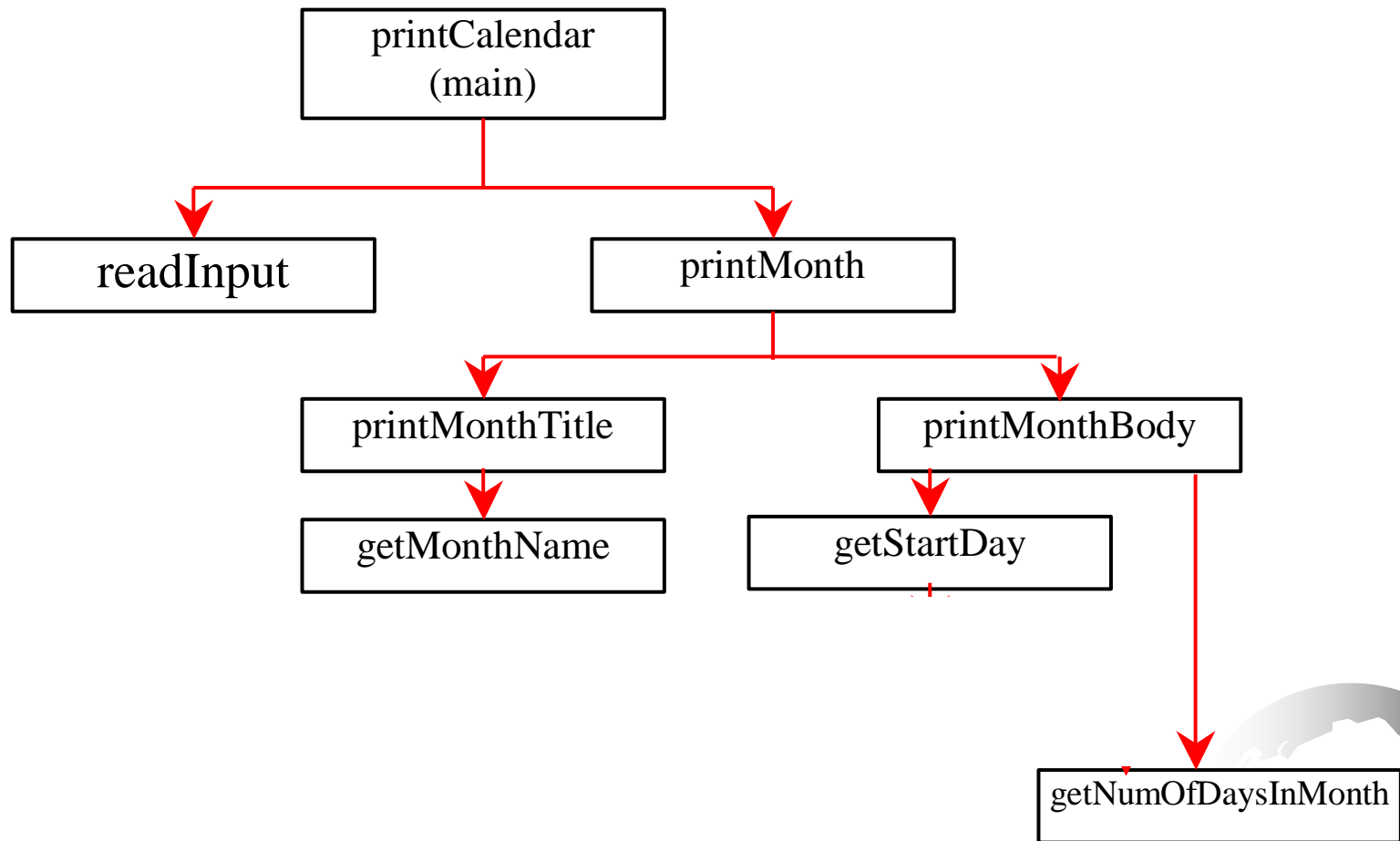


```
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
      April 2009
-----
Sun Mon Tue Wed Thu Fri Sat
      1  2  3  4
 5   6   7   8   9  10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30

C:\book>
```

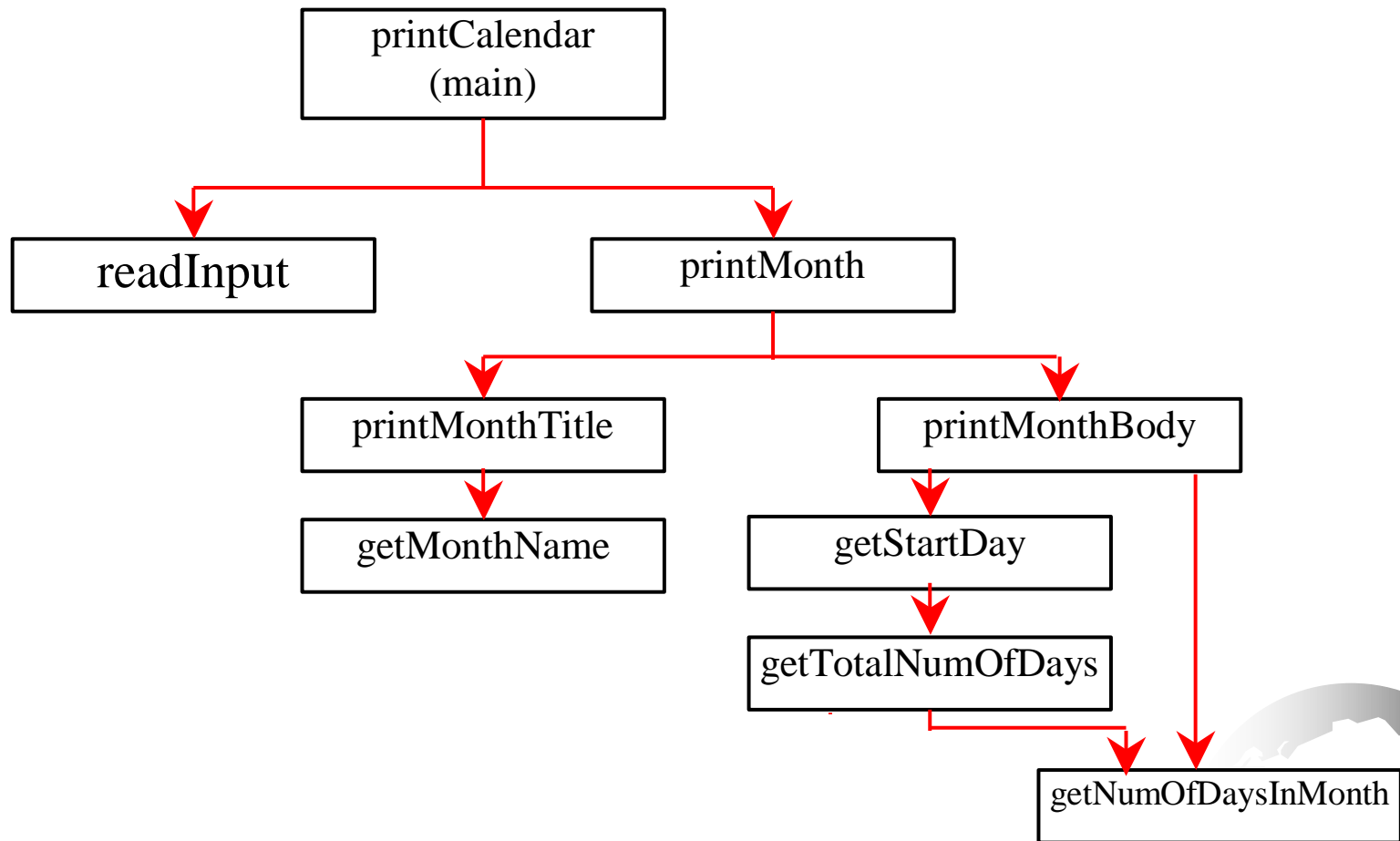


# Design Diagram



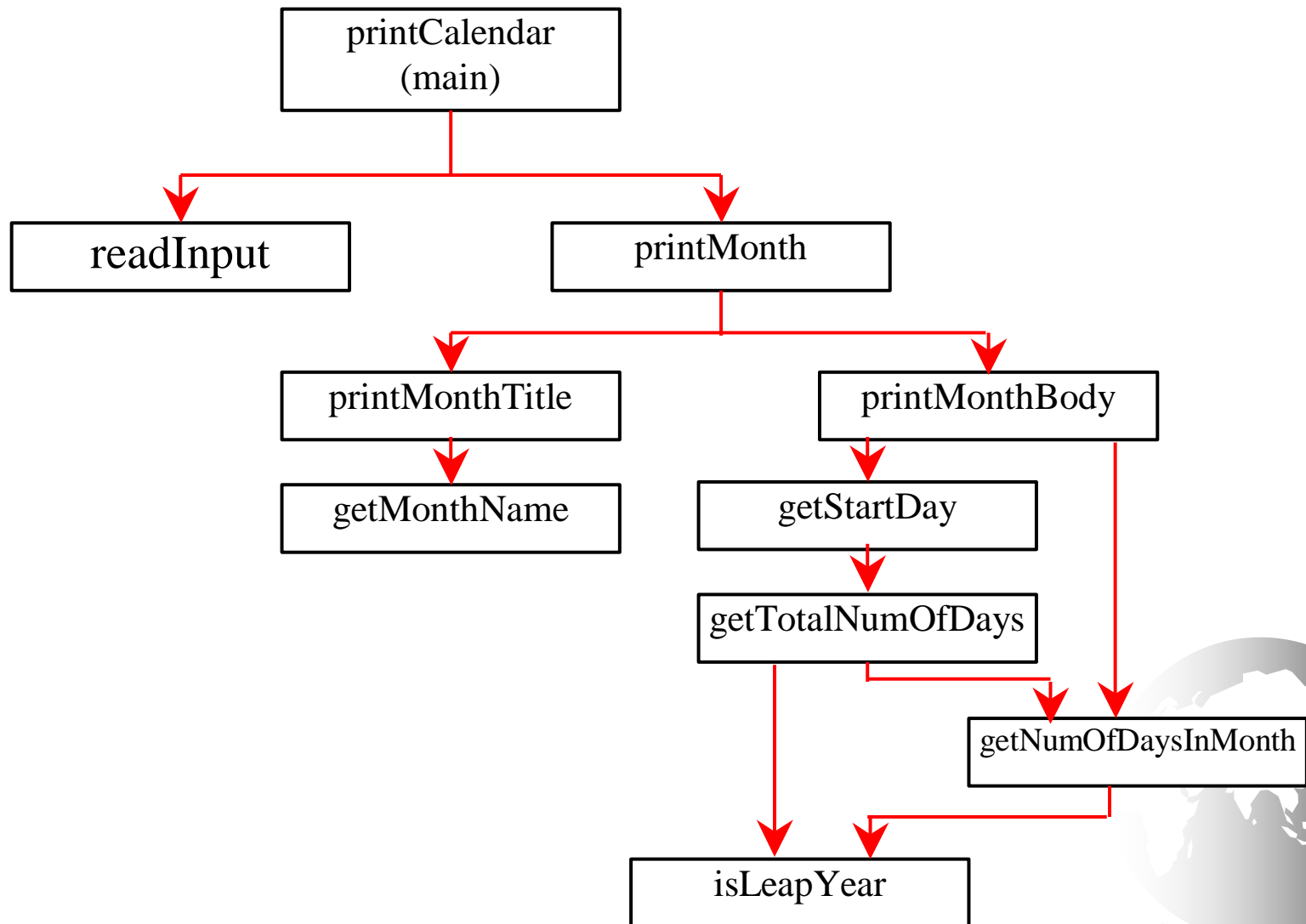


# Design Diagram





# Design Diagram





# Implementation: Top-Down

Top-down approach is to implement one method in the 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 main method first and then use a stub for the printMonth method. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

```
public static boolean isLeapYear(int year) {  
    return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);  
}
```

```
public static boolean isLeapYear(int year) {  
    return true;  
}
```

A Skeleton for printCalendar





# Implementation: Bottom-Up

Bottom-up approach is to implement one method in the 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. Sometimes, they can be used together.

```
public class PrintCalendar {  
    /** Main method */  
    public static void main(String[] args) {  
        System.out.println(isLeapYear(2019));  
    }  
}
```

```
public static boolean isLeapYear(int year) {  
    return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);  
}
```



# ★ Benefits of Stepwise Refinement

**Simpler** Program

**Reusing** Methods

Easier Developing, Debugging, and Testing

Better Facilitating **Teamwork**

