

Lecture #7

CSC 200-04L Fall 2018

Ambrose Lewis
tjl274@email.vccs.edu

Agenda

- Theory of Computation
 - Finite State Machines
 - Turing Machines
 - Big O Notation
- Methods & Parameters (Chapter 6)

Theory of Computation

- Theory of Computation is a branch of computer science that deals with how efficiently problems can be solved via a “model of computation” using an algorithm.
- Model of Computation:
 - A set of allowable operations used in computation and their respective costs
 - By assuming a certain model of computation, it is possible to analyze the computational resources required or to discuss the limitations of algorithms or computers.
 - Abstract representation of computer, focus is on the algorithm
 - Two examples: Finite State Machine & Turing Machine

Theory of Computation:

Finite State Machines

- A Finite State Machine (FSM) is an abstract computer that can be in one of a finite number of states.
- The FSM is in only one state at a time
 - The state it is in at any given time is called the current state.
- The FSM can change from one state to another when initiated by a triggering event or condition; this is called a transition.
- A particular FSM is defined by a list of its states, and the triggering condition for each transition.

Theory of Computation:

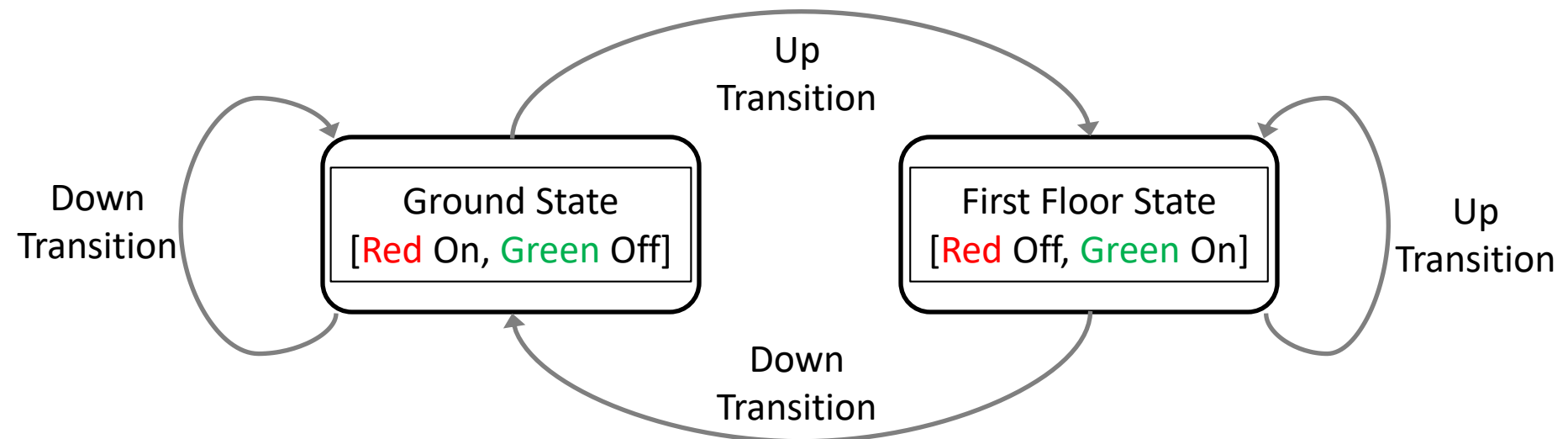
FSM Example

- Step 1: Describe the machine in words
 - In this example, we'll be designing a controller for an elevator.
 - The elevator can be at one of two floors: Ground or First.
 - There is one button that controls the elevator, and it has two values: Up or Down.
 - Also, there are two lights in the elevator that indicate the current floor: Red for Ground, and Green for First.
 - At each time step, the controller checks the current floor and current input, changes floors and lights in the obvious way.

Theory of Computation:

FSM Example

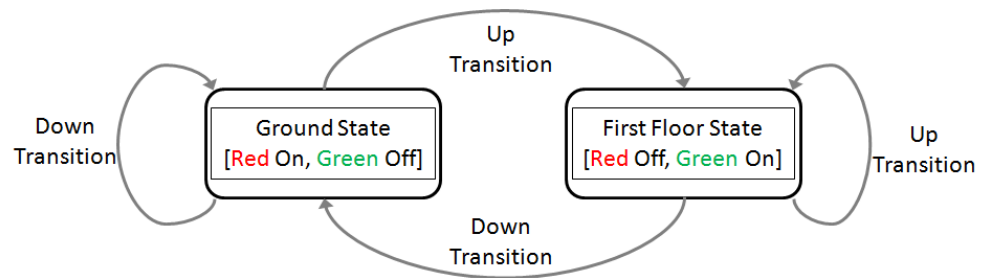
- Step 2: Draw and Label FSM Diagram
 - Rectangles are states
 - Arrows are transitions



Theory of Computation:

FSM Example

- Step 3: Build a FSM Table
 - For each state & input need to define the next state and the condition of the two lights



Current State	Input	Next State	Red Light	Green Light
Ground Floor	Up	First Floor	On	Off
Ground Floor	Down	Ground Floor	On	Off
First Floor	Up	First Floor	Off	On
First Floor	Down	Ground Floor	Off	On

Theory of Computation:

Turing Machines

- A Turing machine is an idealized computing device consisting of:
 - a read/write head (or 'scanner') with a paper tape passing through it.
 - The tape is divided into squares, each square bearing a single symbol--'0' or '1', for example.
 - This tape is the machine's general purpose storage medium, serving both as the input, output, and as a working memory
- The input that is inscribed on the tape before the computation starts must consist of a finite number of symbols.
- The tape is of infinitely long! Turing's aim was to show that there are tasks that these machines are unable to perform, even given unlimited working memory and unlimited time.

Theory of Computation:

Turing Machines Cont...

- Turing Machines have state (similar to FSM)
- Turing Machine have operations:
 - Read the current symbol under the head
 - Write a symbol to the current square (overwrites existing value)
 - move the tape left one square
 - move the tape right one square
 - change state
 - Halt (stop program)
- Turing Machines have instructions
 - Instructions triggered based on current state and the current symbol
 - Take action by performing one of the operations above
- The halting problem tries to determine, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.

Theory of Computation:

Turing Machine Example

- The problem is to set up the machine so that if the scanner is positioned over any square and the machine is set in motion, it will print alternating binary digits on its tape, 0 1 0 1 0 1..., working to the right from its starting place, and leaving a blank square in between each digit.
- In order to do its work the machine makes use of four states, labelled '**a**', '**b**', '**c**' and '**d**'. The machine is in state **a** when it starts work.
- The table of instructions for this machine is below.
- The top line of the table reads: if you are in state **a** and the current square is blank then print 0 on it, move right one square, and go into state **b**.
- Assuming a blank tape at start, will this program ever stop?

state	scanned symbol	print	move	next state
a	blank	0	R	b
b	blank		R	c
c	blank	1	R	d
d	blank		R	a

Theory of Computation:

Three Areas

- Automata Theory: Study of abstract computing machines [like FSM or Turing Machines as discussed]
- Computability Theory: Can a problem be solved on a computer? [the halting problem]
- Computational Complexity [how long?]

Theory of Computation:

Computability Theory

- Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
- The halting problem is historically important because it was one of the first problems to be proved undecidable.
 - Undecidable = impossible to construct a single algorithm that always leads to a correct yes-or-no answer.
- If you're interested in learning more, look up NP Complete problems

Theory of Computation: Computational Complexity

- Considers not only whether a problem can be solved at all on a computer, but also how efficiently the problem can be solved.
- Two major aspects are considered:
 - time complexity: how many steps does it take to perform a computation?
 - Space complexity: how much memory is required to perform that computation?
- Big O Notation is used to understand what happens to the time or space complexity as the input grows
 - This is called Analysis of Algorithms!

Theory of Computation:

Big O Notation

- Shorthand to summarize what happens to the time (or space) need to produce an answer as the input to a program grows
 - For a given input N , determine how many steps/operations are needed to produce the answer...you are concerned with the “order of magnitude” cost in time/space
 - Algorithm/Problem is broken down into logical chunks, the “most expensive” of these drives the time (or space) complexity
- Typically concerned with the worst case, setting an upper bound on the time/space.

Theory of Computation:

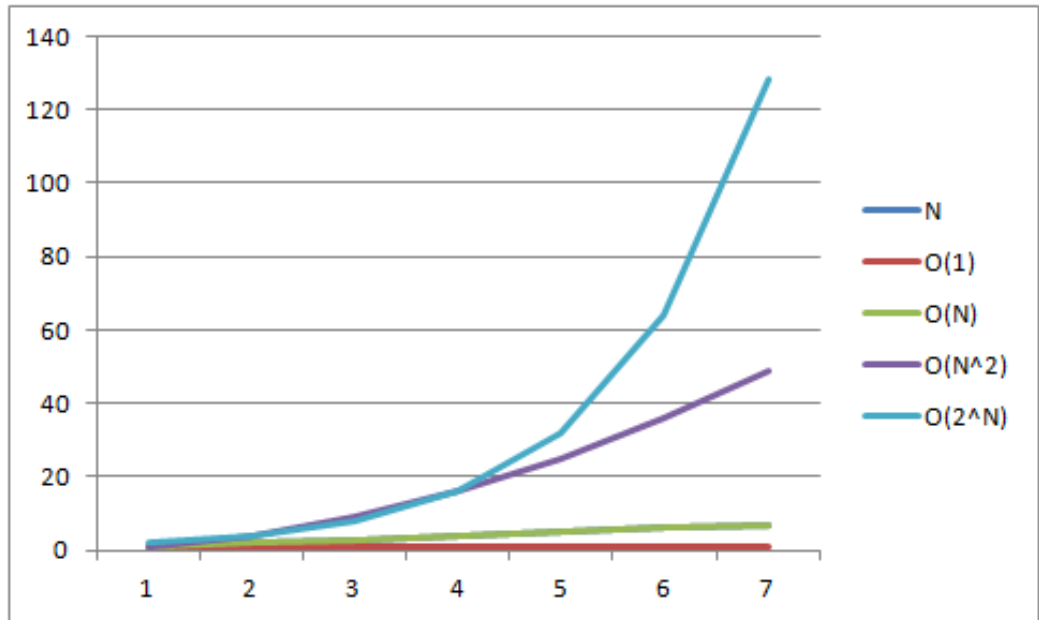
Big O Notation Examples

- Some Examples:
 - $O(1)$ is constant time or “order 1” [stays the same regardless of the input]
 - $O(n)$ is linear time or “order n ” [time to process grows linearly with the size of the input]
 - $O(n^2)$ is quadratic time or “order n squared”
 - $O(\log n)$ is logarithmic time
 - $O(n \log n)$ is “log-linear” time
 - $O(c^n)$ is exponential time (for $c > 1$)
- Why does this matter? Look at graph on next slide

Theory of Computation

Big O Notation Examples

N	O(1)	O(N)	O(N ²)	O(2 ^N)
1	1	1	1	2
2	1	2	4	4
3	1	3	9	8
4	1	4	16	16
5	1	5	25	32
6	1	6	36	64
7	1	7	49	128



What happens when N gets “big”?

N	O(1)	O(N)	O(N ²)	O(2 ^N)
48	1	48	2304	281,474,976,710,656.00
49	1	49	2401	562,949,953,421,312.00
50	1	50	2500	1,125,899,906,842,620.00

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;  
}
```

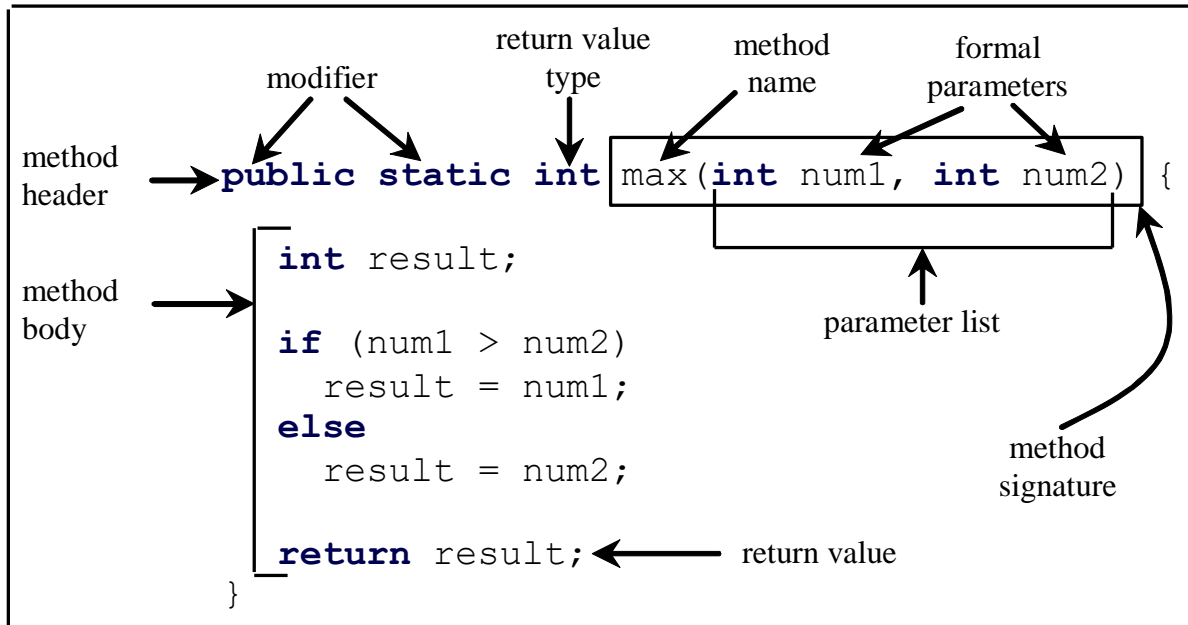
Invoke a method

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

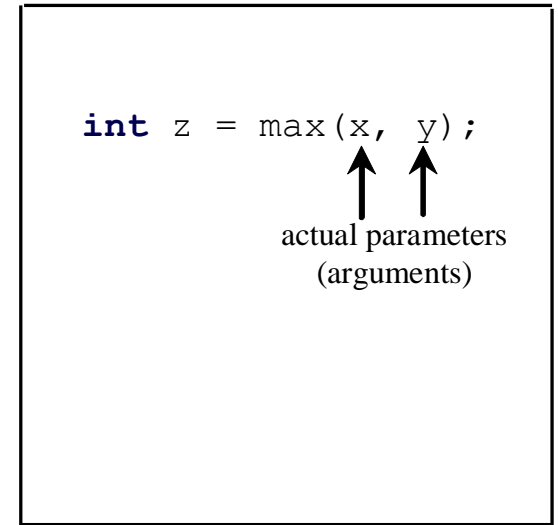
Defining Methods

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

Define a method



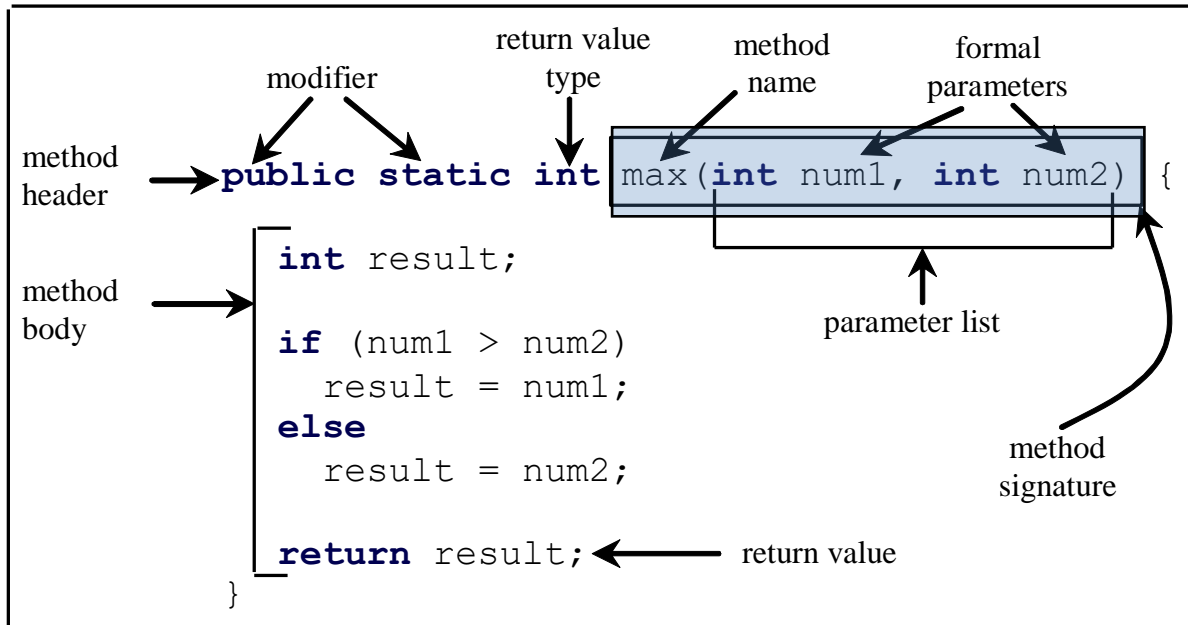
Invoke a method



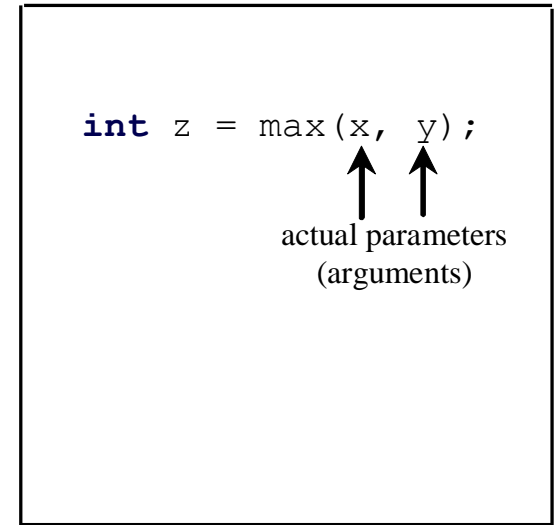
Method Signature

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

Define a method



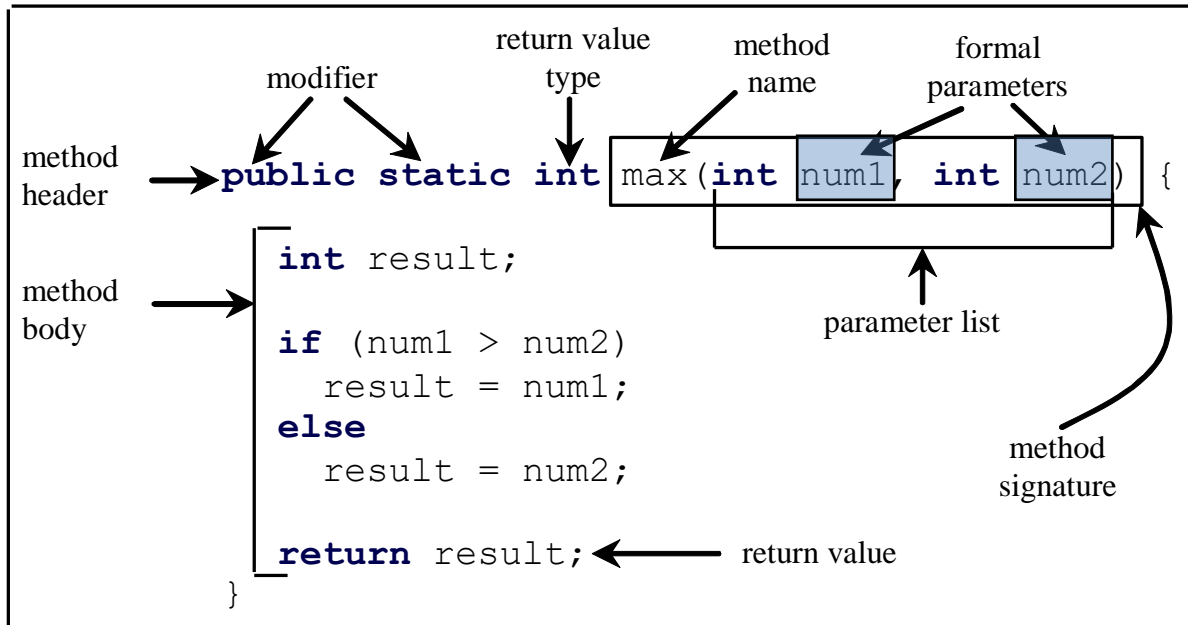
Invoke a method



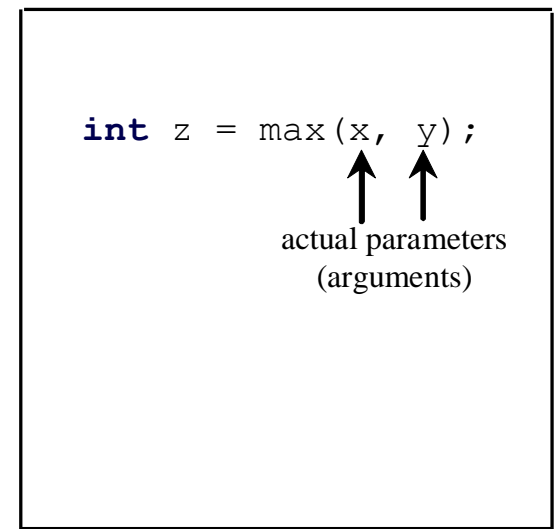
Formal Parameters

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

Define a method



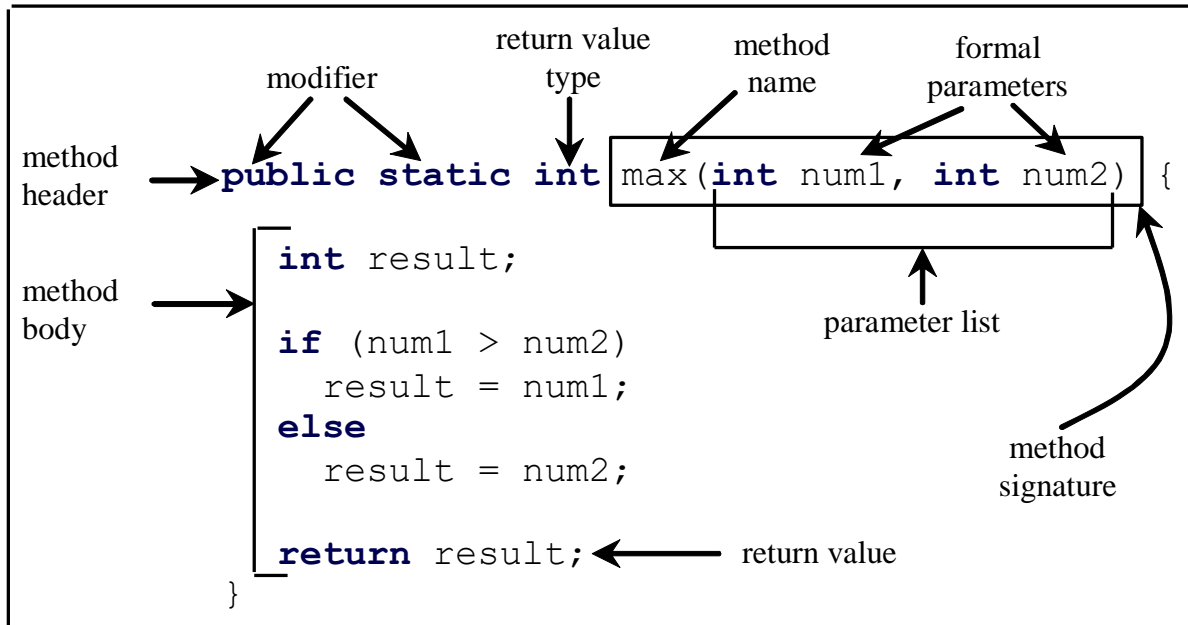
Invoke a method



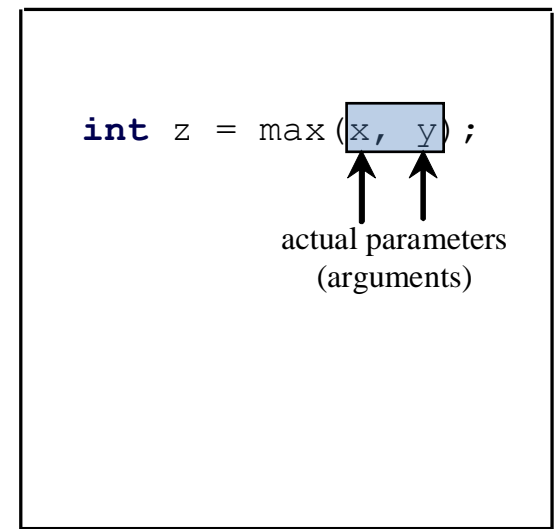
Actual Parameters

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



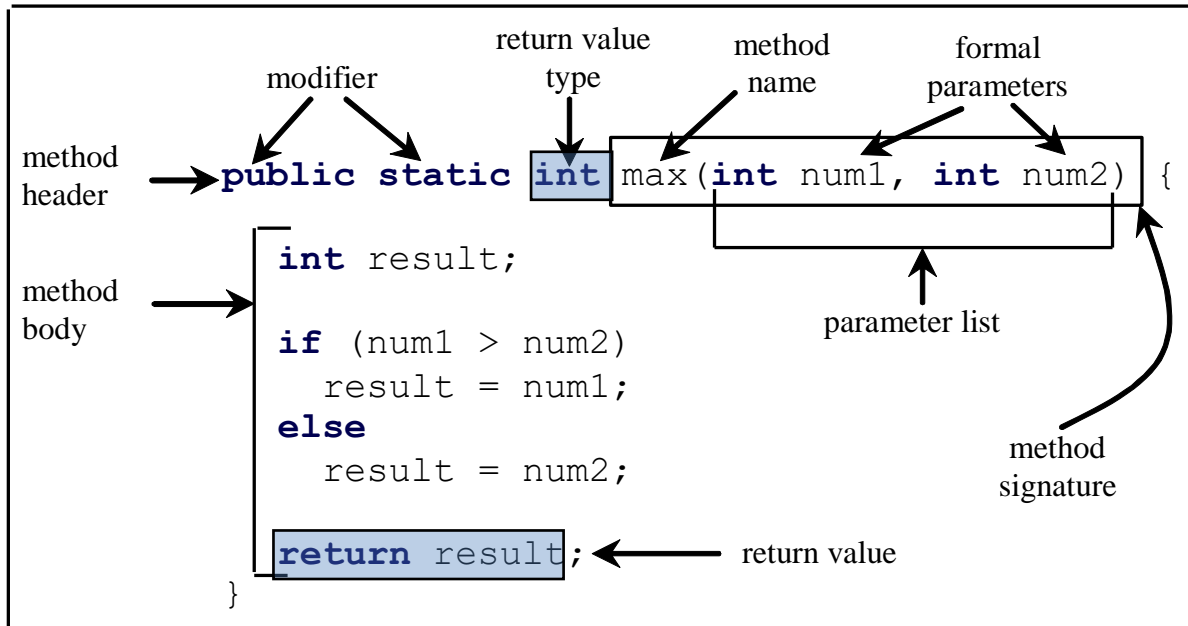
Invoke a method



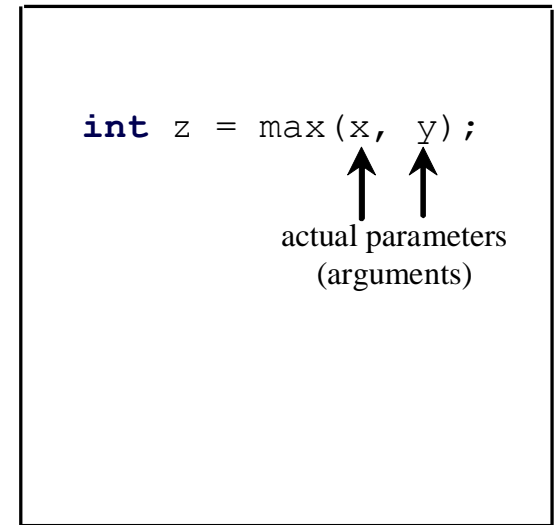
Return Value Type

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



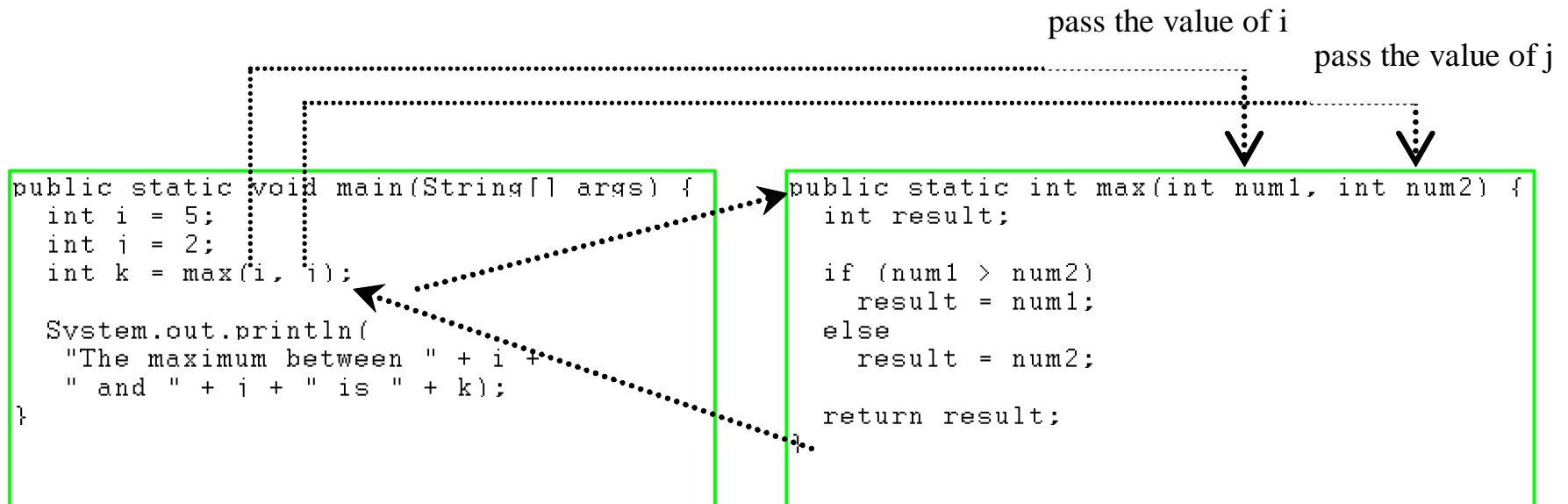
Calling Methods

Testing the `max` method

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

```
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 between " + i + " and " + j + " is " + k);  
    }  
    /** Return the max between two numbers */  
    public static int max(int num1, int num2) {  
        int result;  
        if (num1 > num2)  
            result = num1;  
        else  
            result = num2;  
        return result;  
    }  
}
```

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 i = 2;  
    int k = max(i, i);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + i + " 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.

```
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;  
}
```

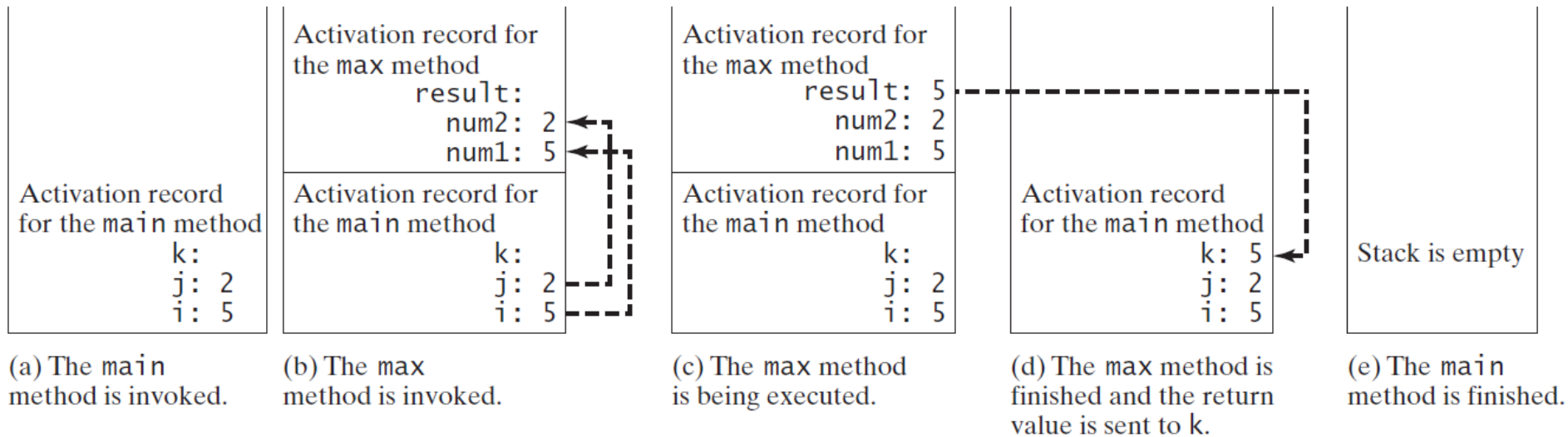
(b)

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 i = 2;  
    int k = max(i, i);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + i + " is " + k);  
}
```

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



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

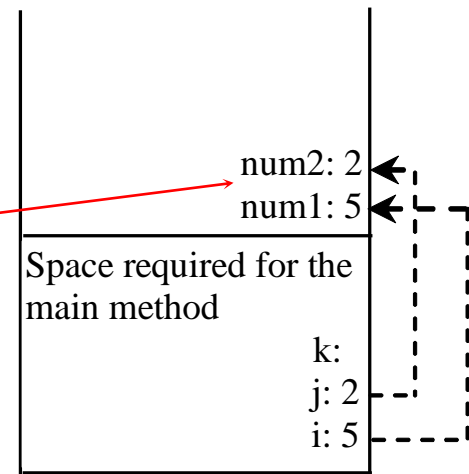
The main method
is invoked.

Trace Call Stack

```
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);  
}
```

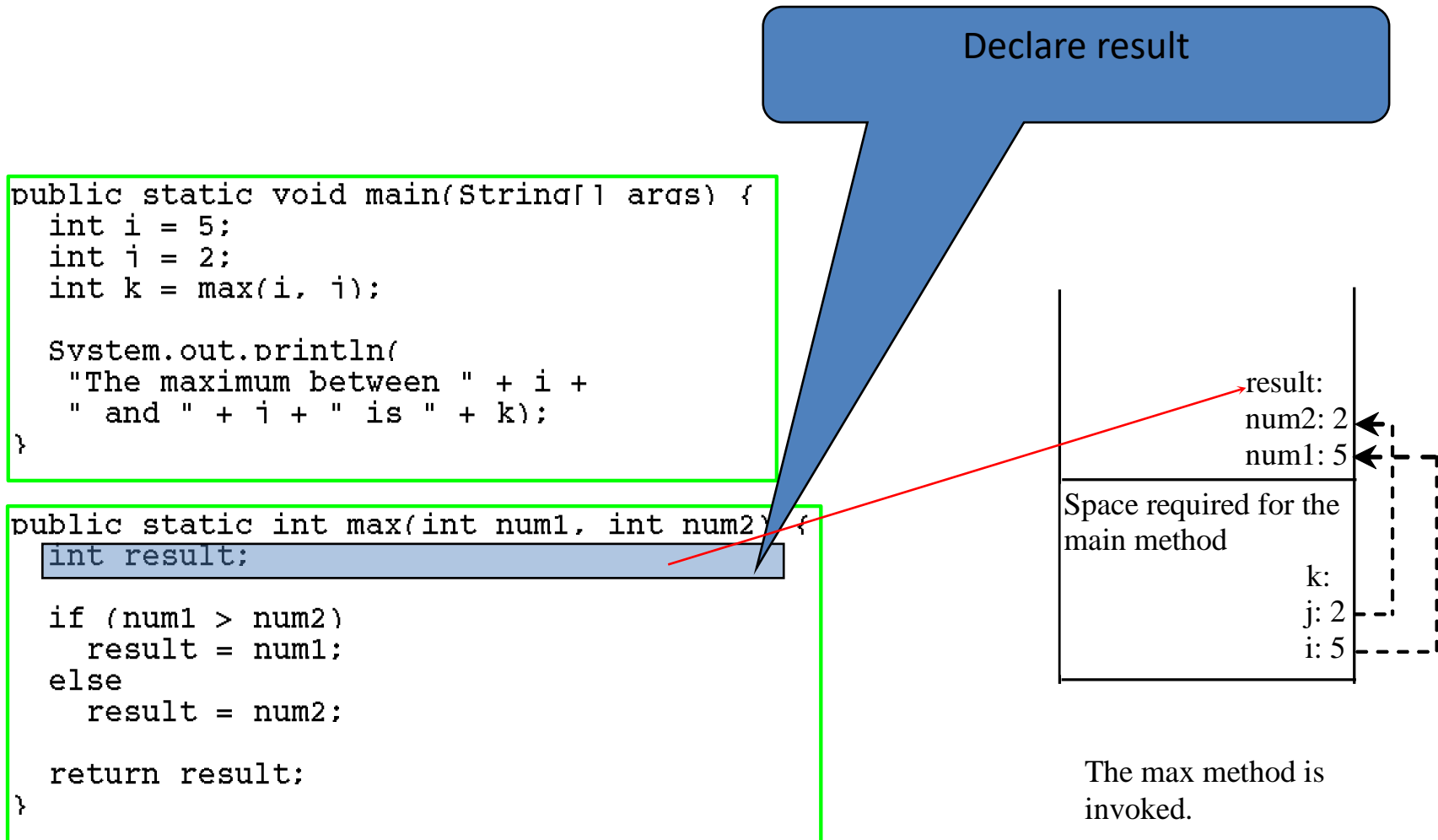
```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

pass the values of i and j to num1
and num2



The max method is
invoked.

Trace Call Stack

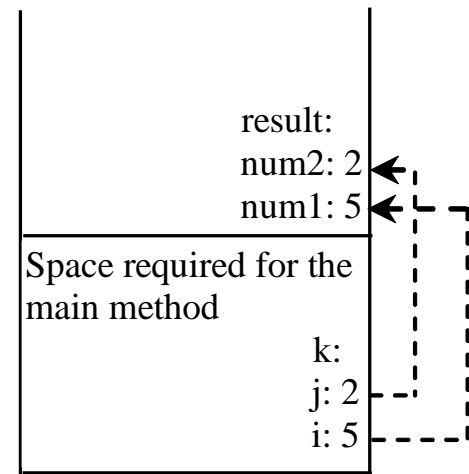


Trace Call Stack

(num1 > num2) is true

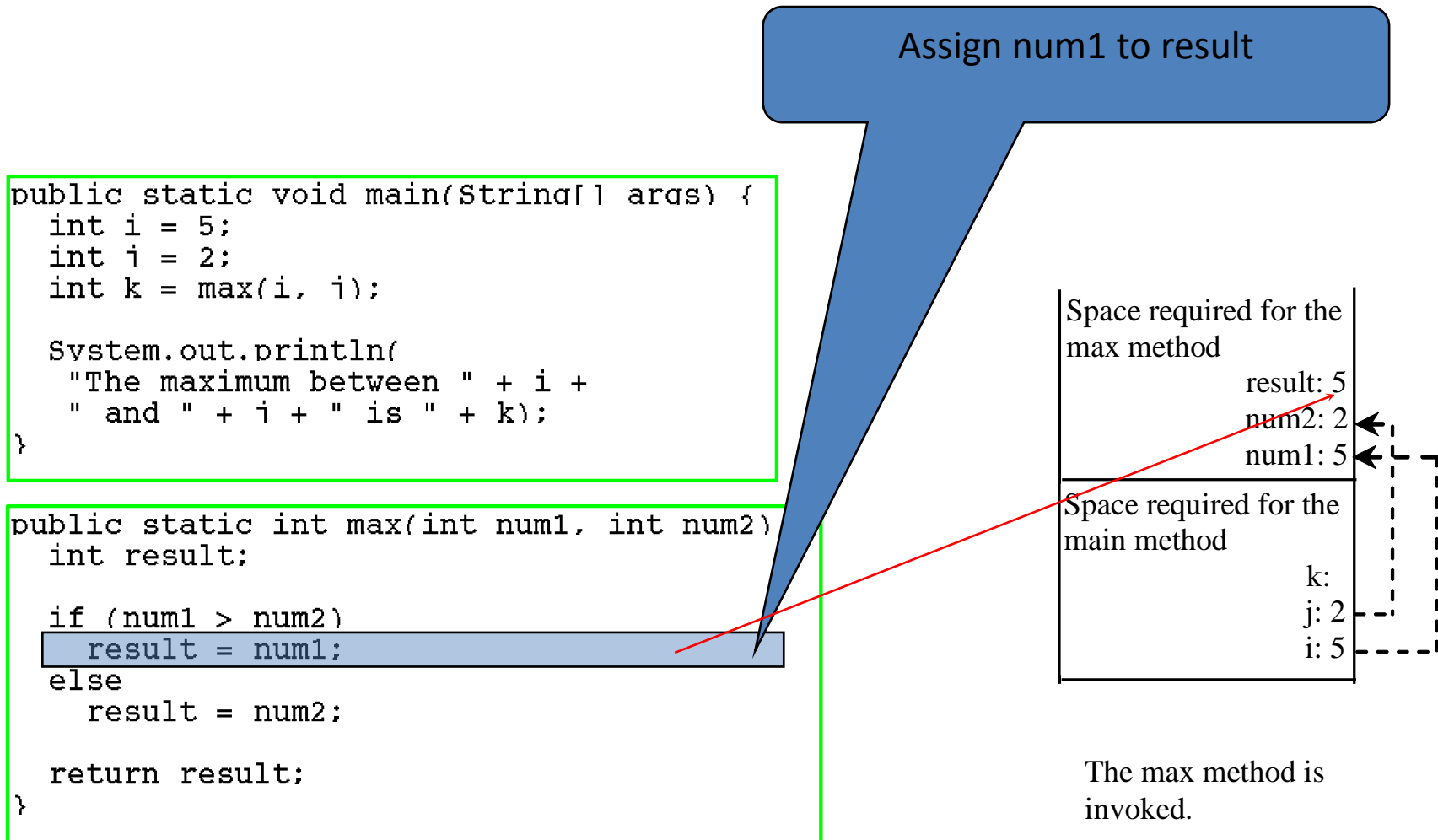
```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

```
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

Return result and assign it 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;  
}
```

Space required for the
max method

result: 5
num2: 2
num1: 5

Space required for the
main method

k: 5
j: 2
i: 5

The max method is
invoked.

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 (does work for you!) but doesn't need to return a value

TestVoid.java

- Write a program that defines a method named `printGrade` that takes in a double variable and prints out the following messages
 - Grade 90.0 or larger, output “A”
 - Grade greater than or equal to 80.0 and less than 90.0, output “B”
 - ...
- Call the method from your main method with inputs of 89.8, 90.01, 33.333, and 76.2

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 Example 1

This program demonstrates passing values to the methods.

```
public class Increment {  
    public static void main(String[] args) {  
        int x = 1;  
        System.out.println("Before the call, x is " + x);  
        increment(x);  
        System.out.println("after the call, x is " + x);  
    }  
    public static void increment(int n) {  
        n++;  
        System.out.println("n inside the method is " + n);  
    }  
}
```

Pass by Value Example 2

```
public class TestPassByValue {
    /** Main method */
    public static void main(String[] args) {
        // Declare and initialize variables
        int num1 = 1;
        int num2 = 2;

        System.out.println("Before invoking the swap method, num1 is " +
            num1 + " and num2 is " + num2);

        // Invoke the swap method to attempt to swap two variables
        swap(num1, num2);

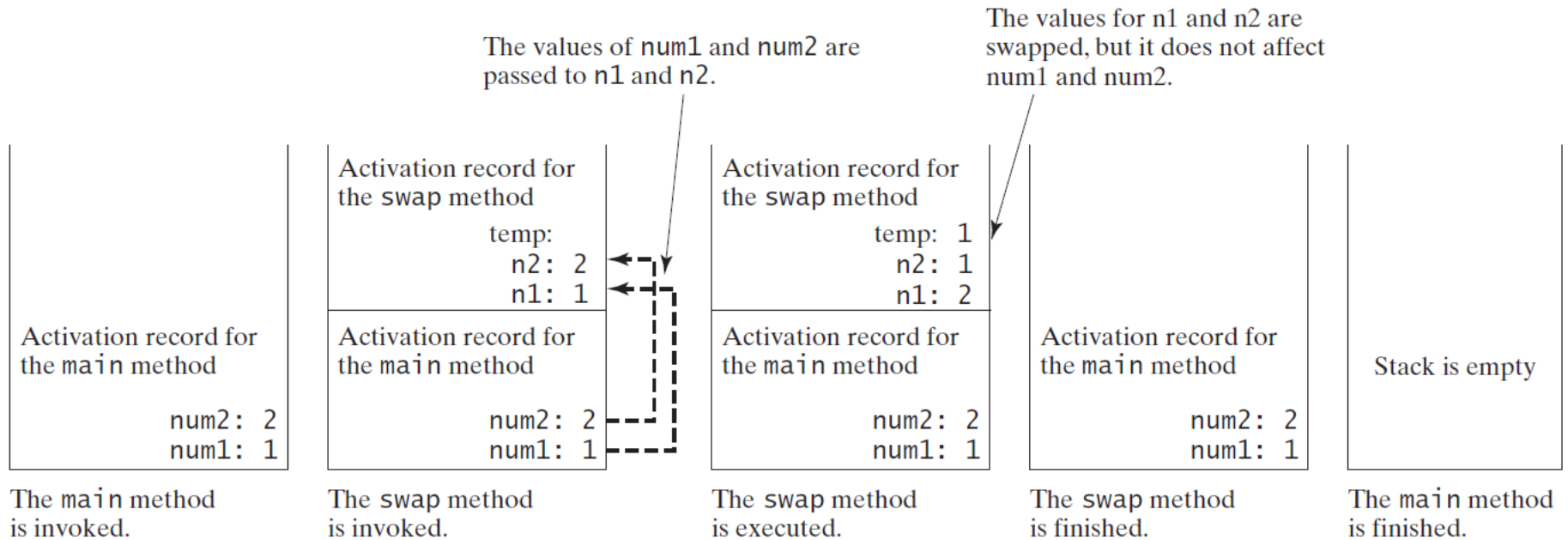
        System.out.println("After invoking the swap method, num1 is " +
            num1 + " and num2 is " + num2);
    }

    /** Swap two variables */
    public static void swap(int n1, int n2) {
        System.out.println("\t\tInside the swap method");
        System.out.println("\t\t\tBefore swapping, n1 is " + n1
            + " and n2 is " + n2);

        // Swap n1 with n2
        int temp = n1;
        n1 = n2;
        n2 = temp;

        System.out.println("\t\t\tAfter swapping, n1 is " + n1
            + " and n2 is " + n2);
    }
}
```

Pass by Value Example 2 Call Stack



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.

Case Study: Converting Hexadecimals to Decimals

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

ABCD =>

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

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

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

Overloading Methods

- Next slide shows example code for overloading a method
- A method is overloaded if it has the same name as another method
- The method name, type, and order of the arguments make up the method's signature
- You can't have methods with the same signature

```

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;  
    }  
  
    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.

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 →

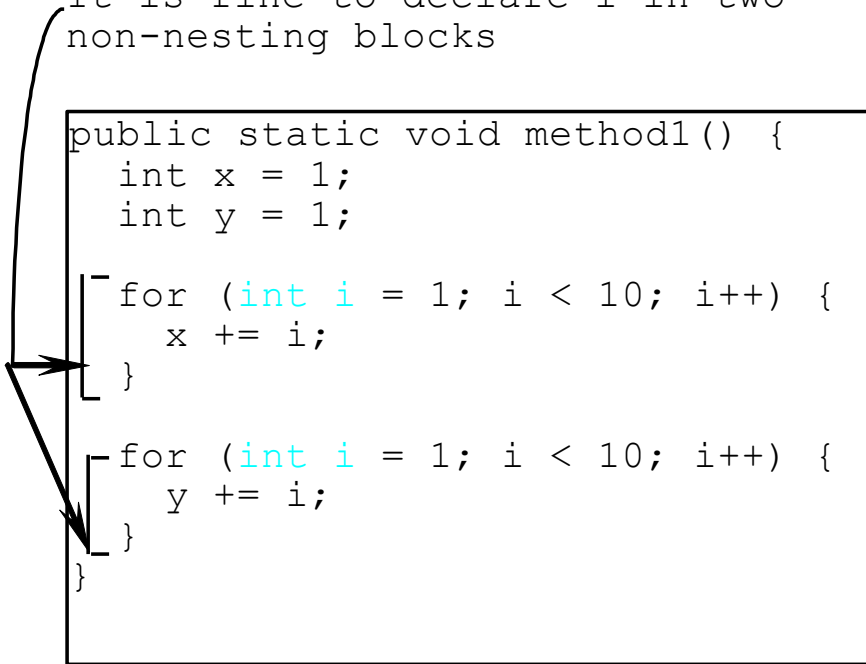
The scope of j →



Scope of Local Variables, cont.

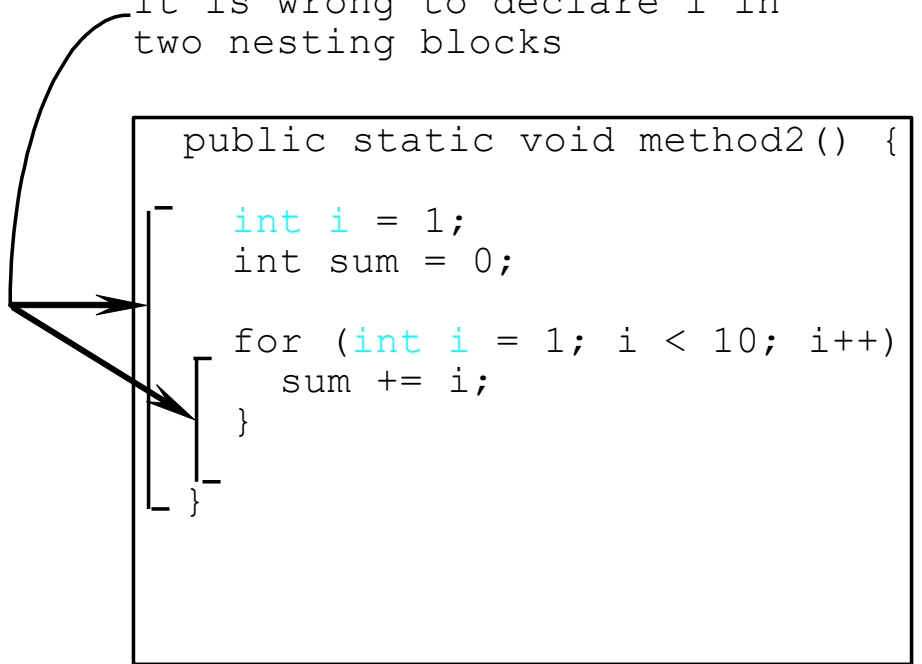
It is fine to declare `i` in two non-nesting blocks

```
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;
    }
}
```

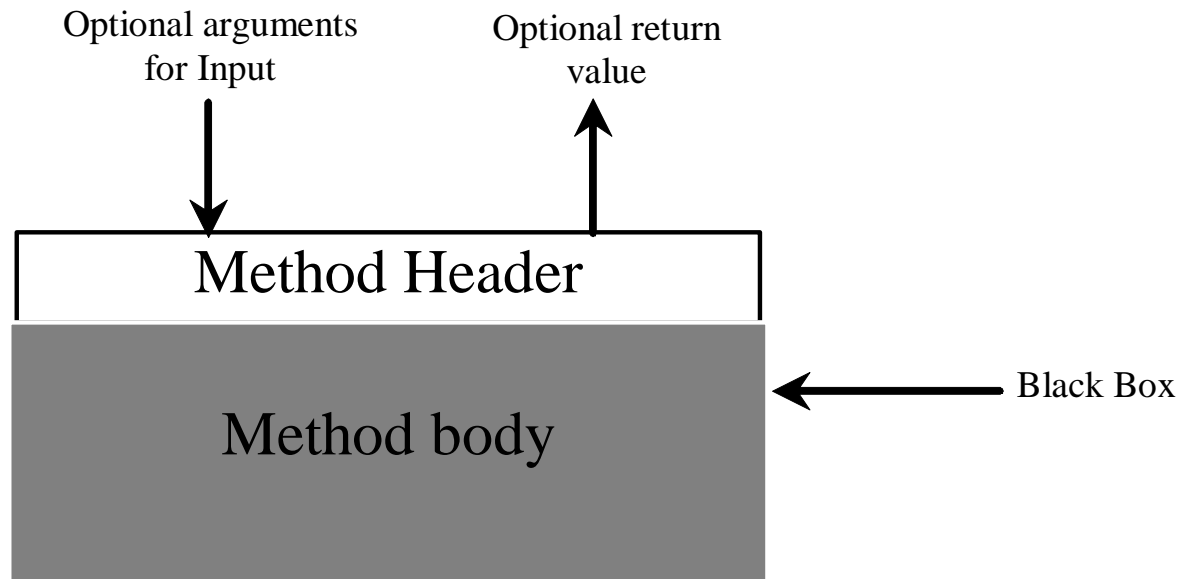
Scope of Local Variables, cont.

// 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.

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))
```

NOTE: This example from the author uses `Math.random`. We will work up an example using `java.util.Random` in class today (posted to Blackboard)

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:

$$'a' + \text{Math.random()} * ('z' - 'a' + 1)$$

So a random lowercase letter is

$$(\text{char})('a' + \text{Math.random()} * ('z' - 'a' + 1))$$

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');
    }
}
```

```

/** Main method */
public static void main(String args[]) {
    final int NUMBER_OF_CHARS = 175;
    final int CHARS_PER_LINE = 25;

    // Print random char between 'a' and 'z'
    // 25 chars per line
    for (int i = 0; i < NUMBER_OF_CHARS; i++) {
        char ch = getRandomLowerCaseLetter();
        if ((i + 1) % CHARS_PER_LINE == 0)
            System.out.println(ch);
        else
            System.out.print(ch);
    }
}

```

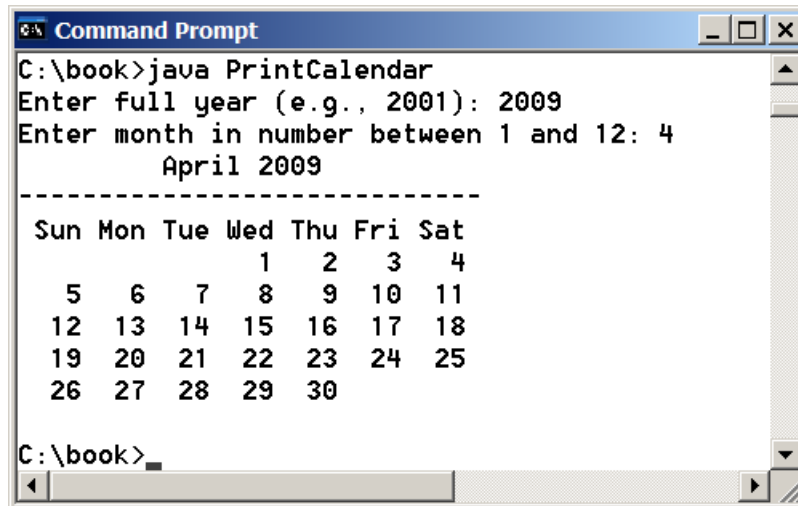
Add this main method to the previous listing to print 175 random lower case characters, 25 per line

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.

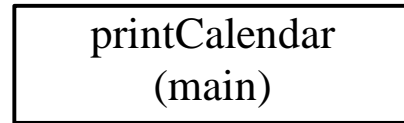
PrintCalendar Case Study

Let us use the PrintCalendar example to demonstrate the stepwise refinement approach. Let's focus on breaking down the program into methods and not implementing these methods...Full code and a "skeleton" version are posted to Blackboard...

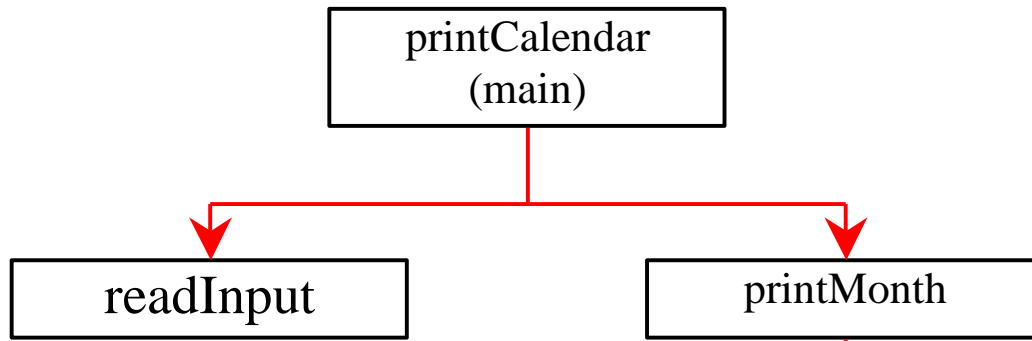


```
Command Prompt
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
    April 2009
-----
Sun Mon Tue Wed Thu Fri Sat
      1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
 26 27 28 29 30
C:\book>
```

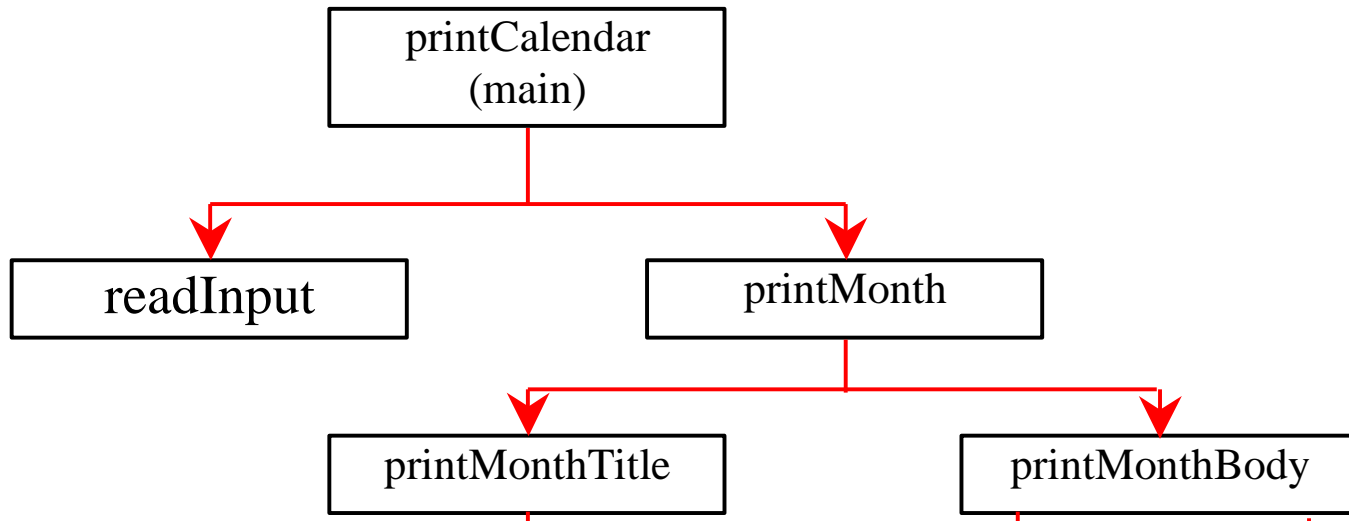
Design Diagram



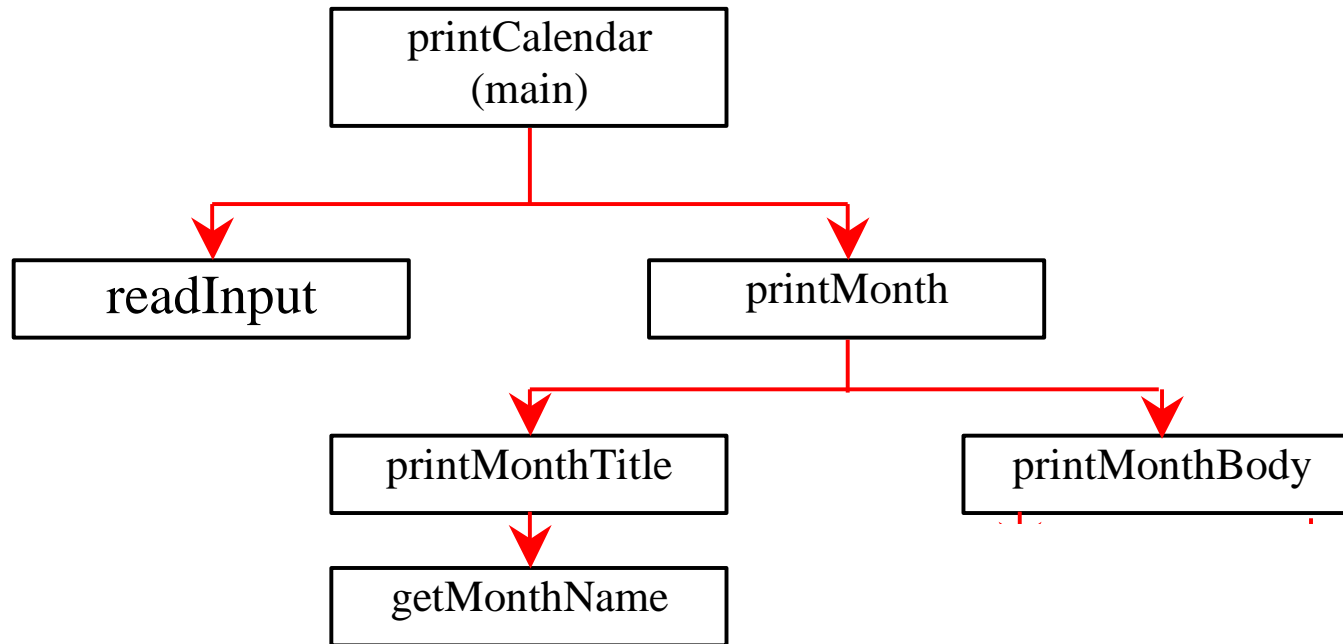
Design Diagram



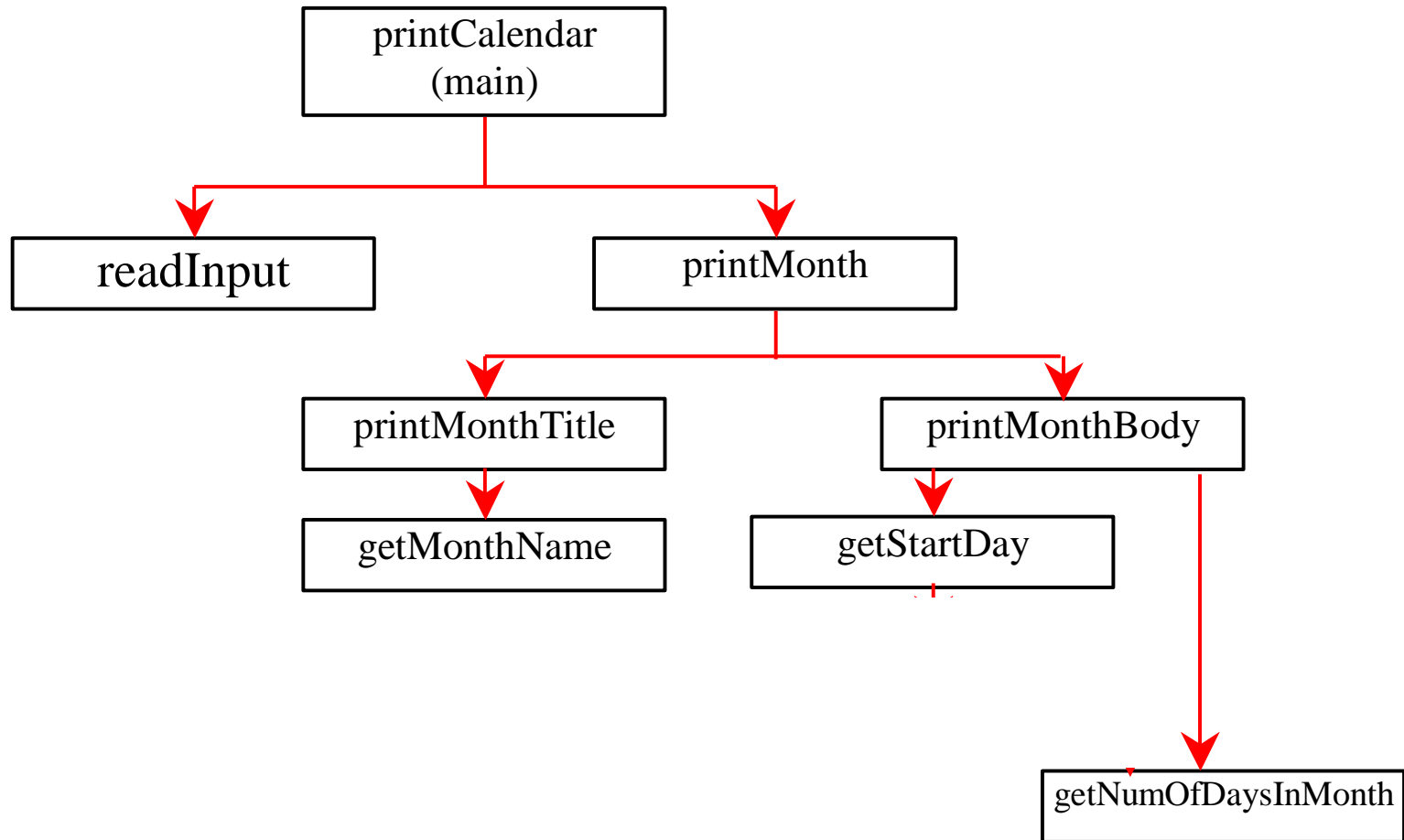
Design Diagram



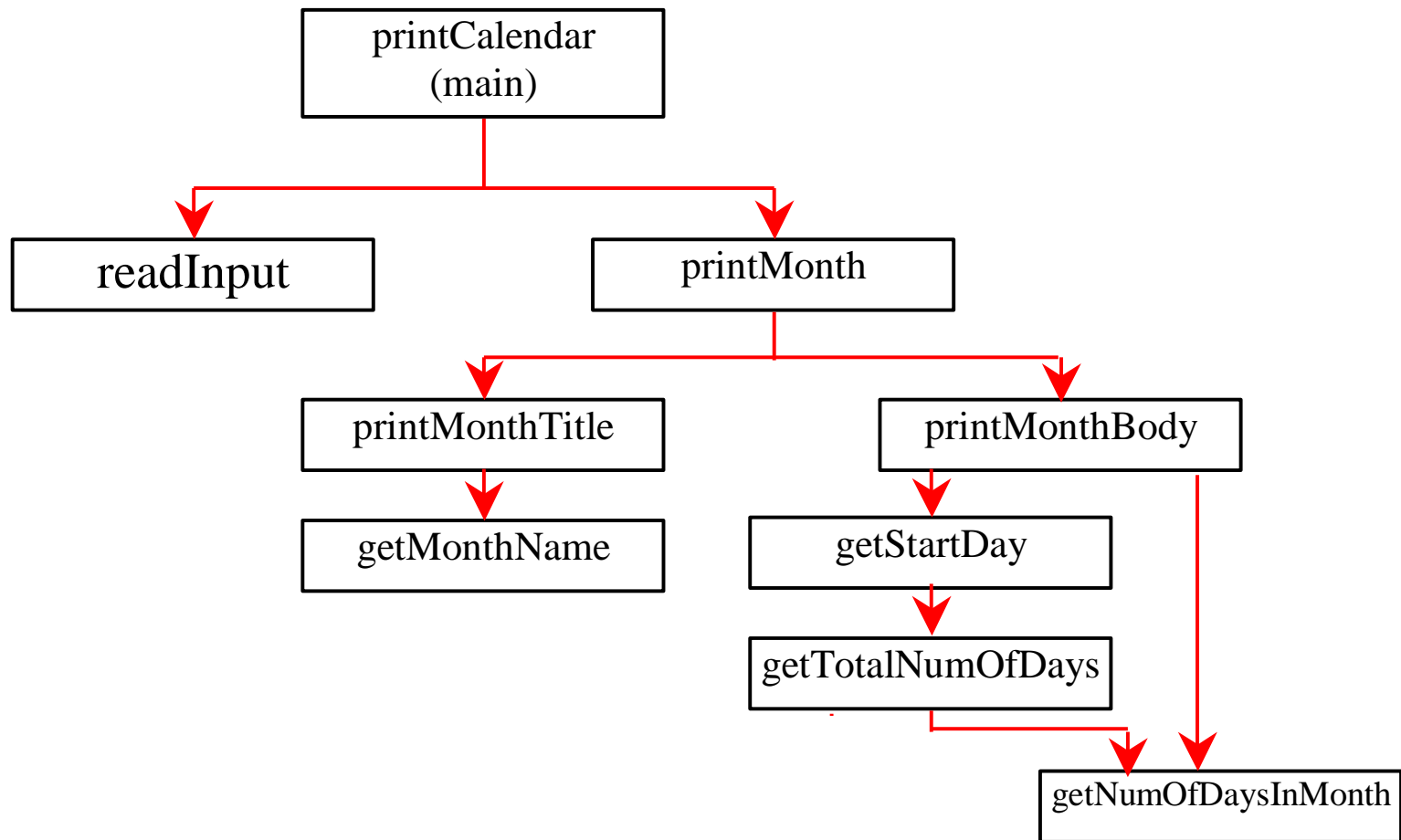
Design Diagram



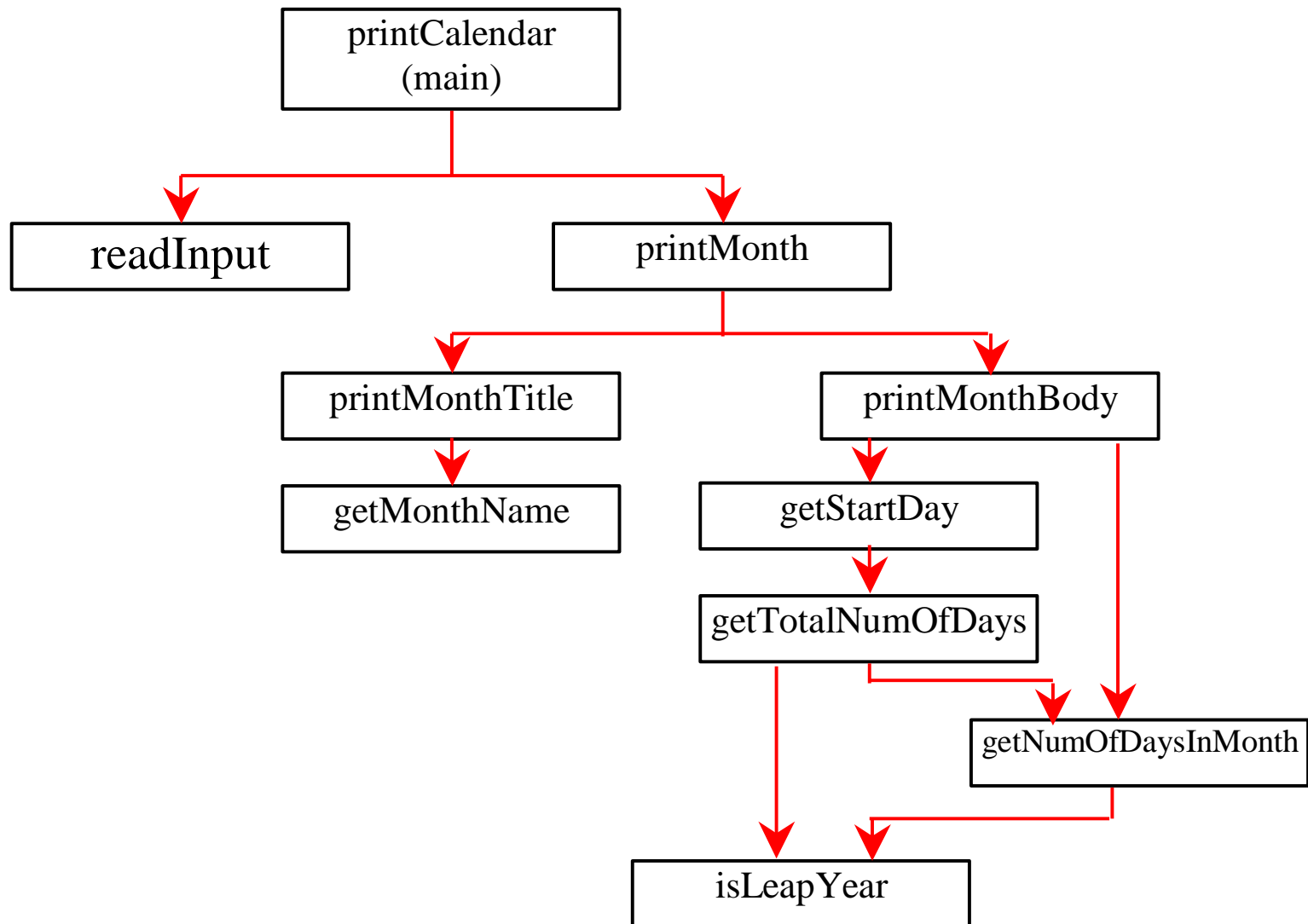
Design Diagram



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:

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.

Benefits of Stepwise Refinement

Simpler Program

Reusing Methods

Easier Developing, Debugging, and Testing

Better Facilitating Teamwork