Chapter 4 Methods

A Problem: Sum Calculation

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

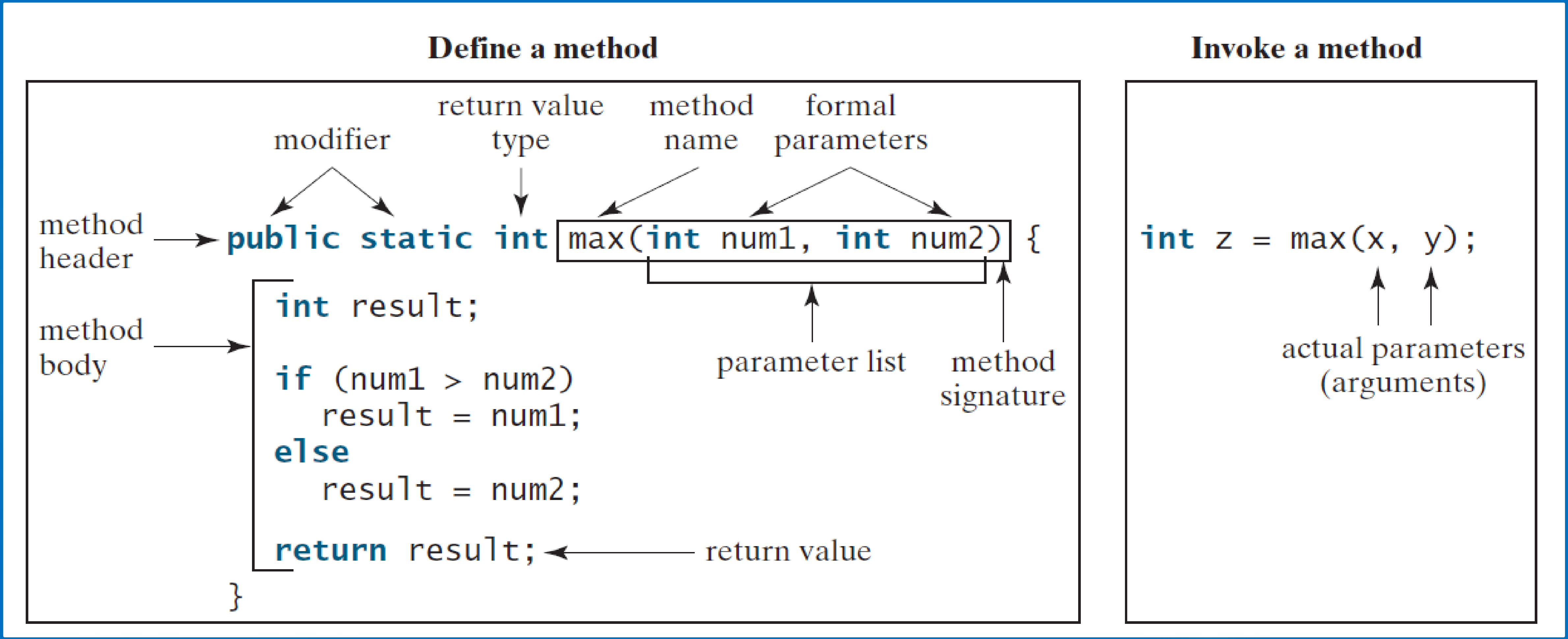
```java
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

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

A Problem: Sum Calculation

```
public class Test{
    public static void main(String[] args) {
        System.out.println(sum(1, 10));
        System.out.println(sum(20, 30));
        System.out.println(sum(35, 45));
    }
    public static int sum(int a, int b) {
        int sum = 0;
        for (int i = a; i <= b; i++)
        sum += i;
        return sum;
    }
}
```

Method Defining

A *method* definition consists of its *method name*, *parameters*, *return value type*, and *body*.



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);
```

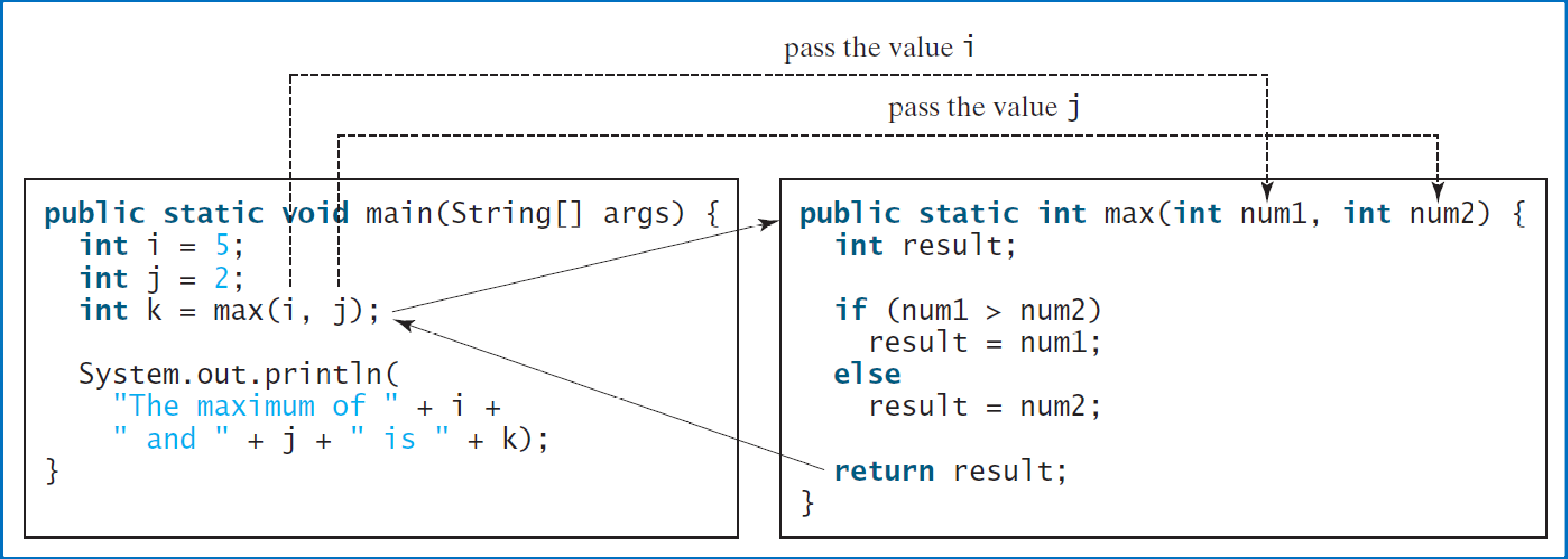
Method Invoking

To execute the method, you have to *call* or *invoke* it. If a method *returns* a value, a call to the method is usually *treated as a value*. If a method returns ***void***, a call to the method must be a *statement*. (Note: A *value-returning method* can also be invoked as a *statement* in Java.)

```java
// as a value
int larger = max(3, 4);
System.out.println(max(3, 4));
// as a statement
System.out.println("Welcome to Java!");
max(3, 4);
```

Method Invoking

When the ***max*** method is invoked, the flow of control transfers to it. Once the ***max*** method is finished, it *returns control back* to the caller.



pass the value i

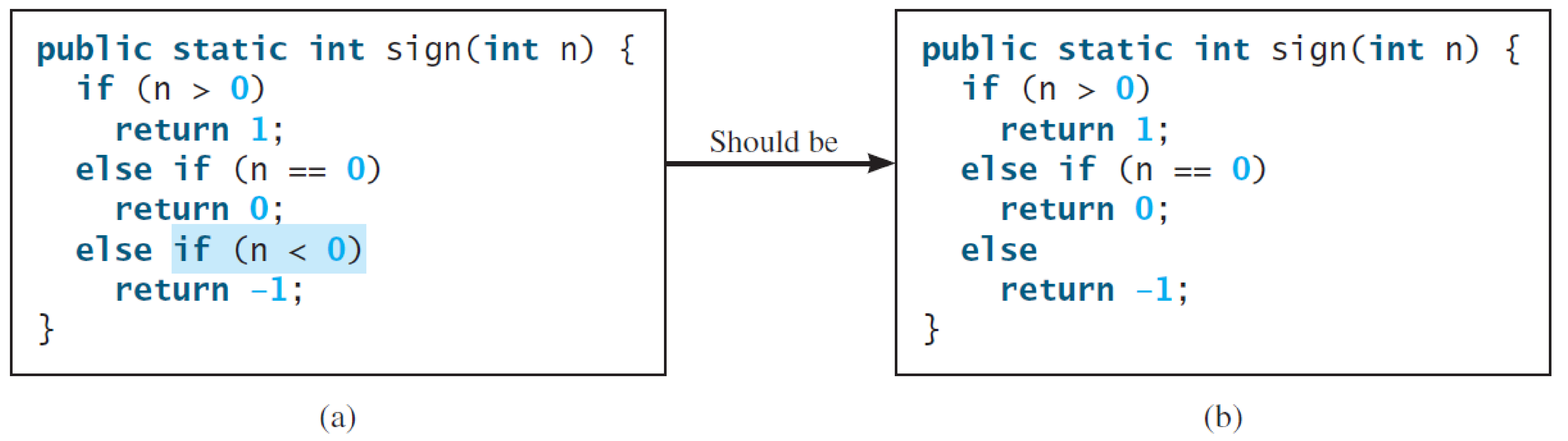
pass the value j

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

  System.out.println(
    "The maximum of " + 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;
}
```

Method Invoking

A *return* statement is required for a value-returning method. The method shown below in (a) is *logically correct*, but it has a compile error because the Java *compiler thinks* that this method might not return a value.



```java
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

```java
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

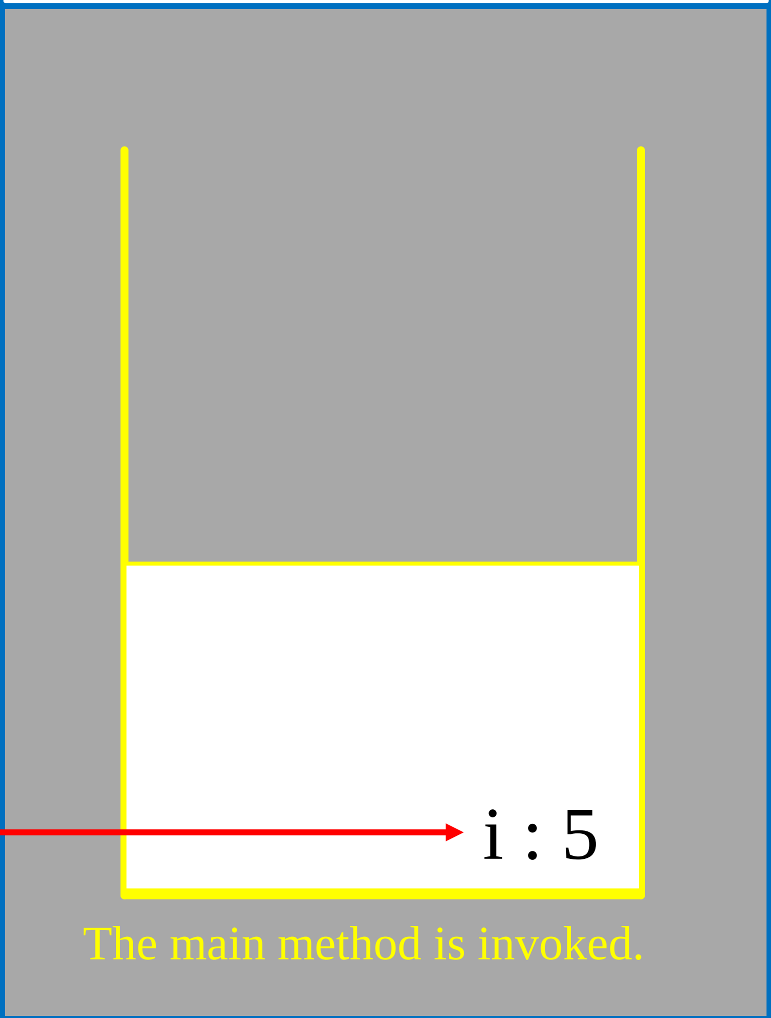
Call Stack

Each time a method is invoked, the system creates an *activation record* (also called an *activation frame*) that *stores parameters and variables* for the method and places the *activation record* in an area of *memory* known as a *call stack*. A call stack is also known as an *execution stack*, *runtime stack*, or *machine stack*, and it is often shortened to just *“the stack”*. When a method calls another method, the *caller’s* activation record is *kept intact*, and a new *activation record* is created for the new method called. When a method *finishes* its work and *returns to its caller*, its *activation record* is *removed* from the call stack.

Call Stack

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

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

i is declared and initialized

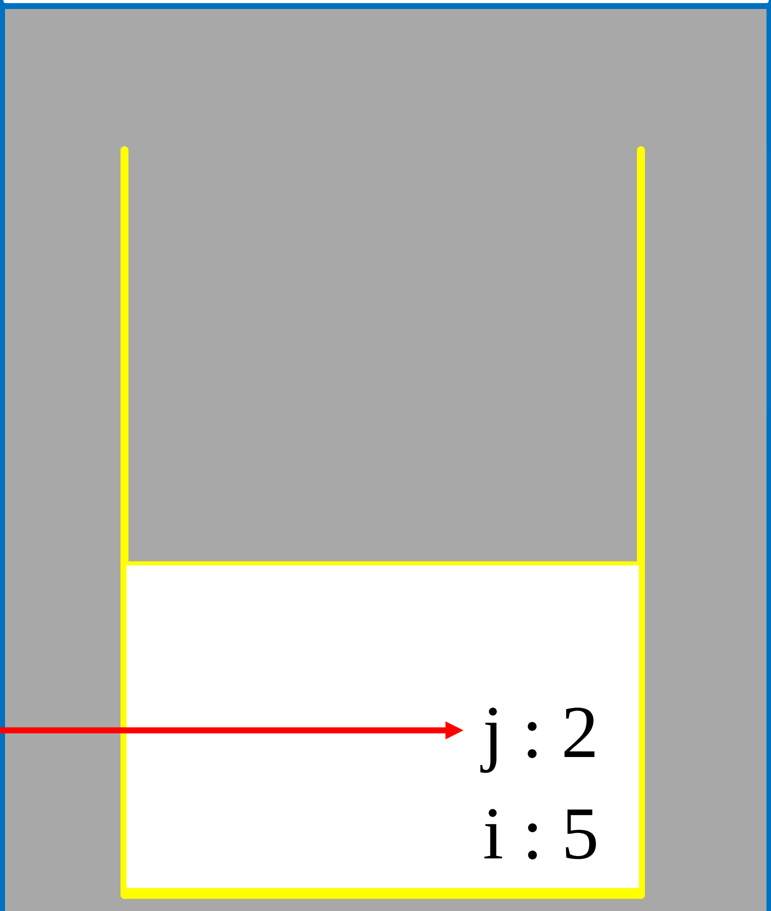


The main method is invoked.

Call Stack

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

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

j is declared and initialized

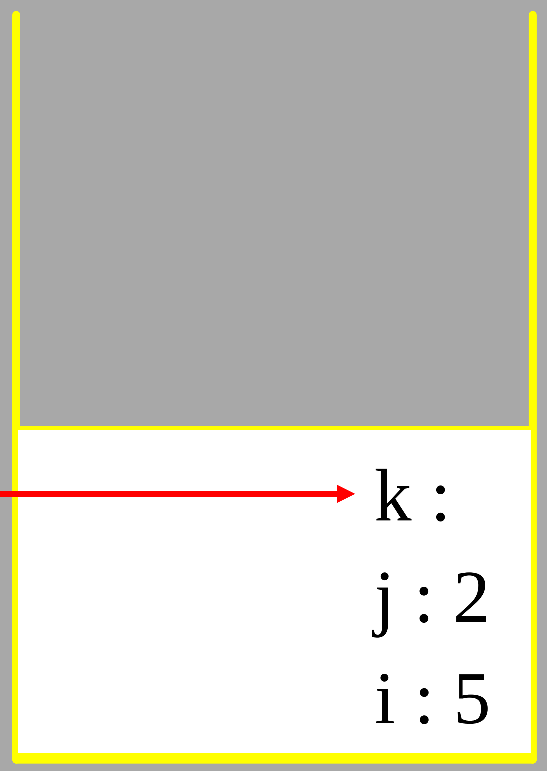


The main method is invoked.

Call Stack

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

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

k is declared

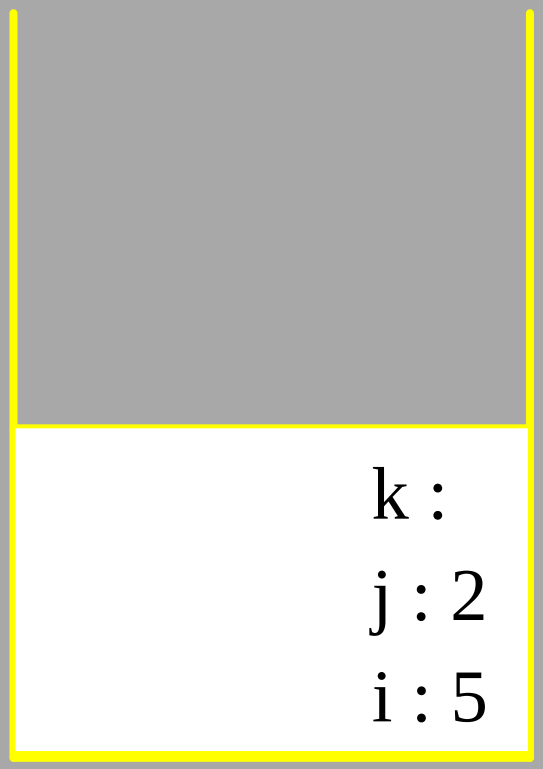


The main method is invoked.

Call Stack

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

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

invoke max(i,j)

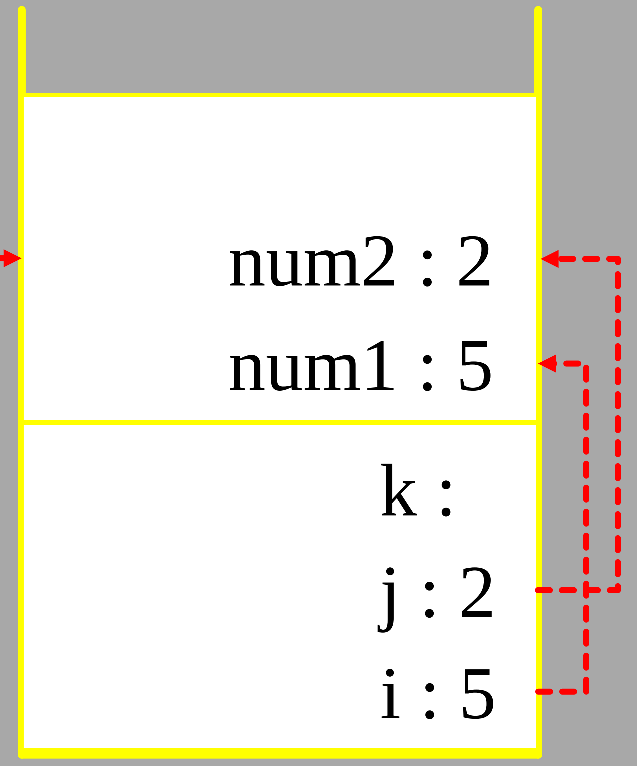


The main method is invoked.

Call Stack

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

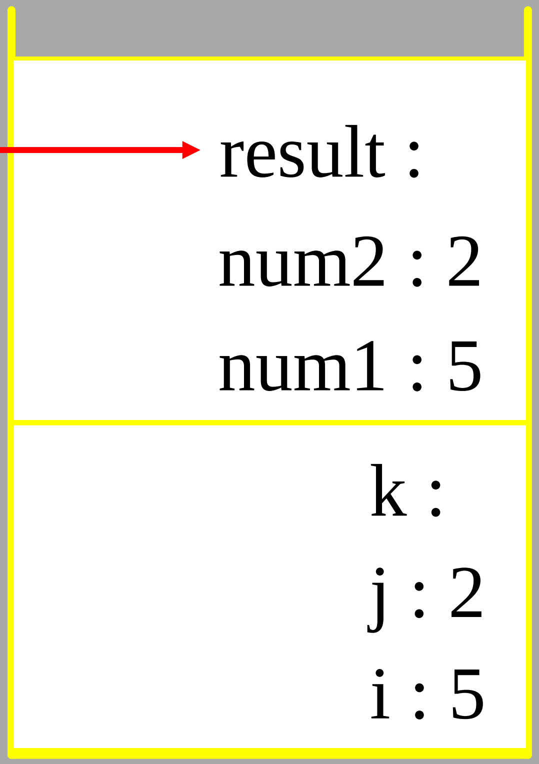


The max method is invoked.

Call Stack

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

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

result is declared

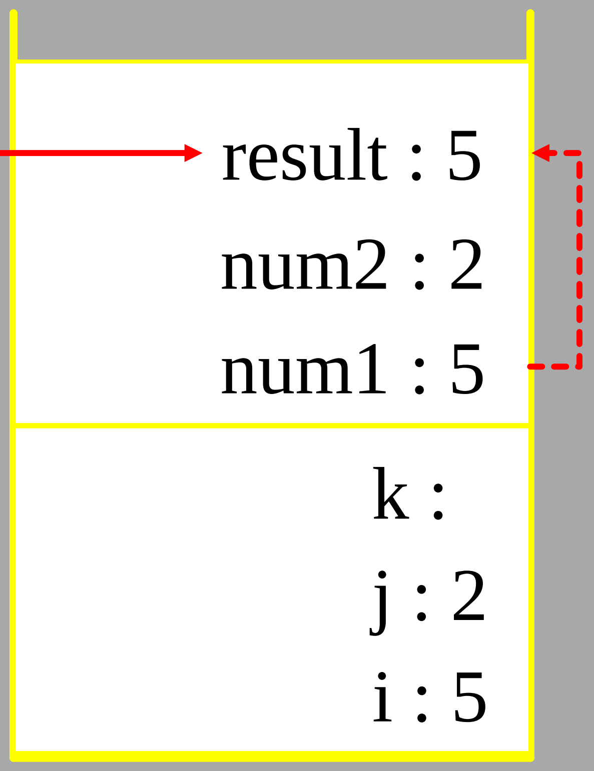


The max method is invoked.

Call Stack

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

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

result is assigned



result : 5

num2 : 2

num1 : 5

k :

j : 2

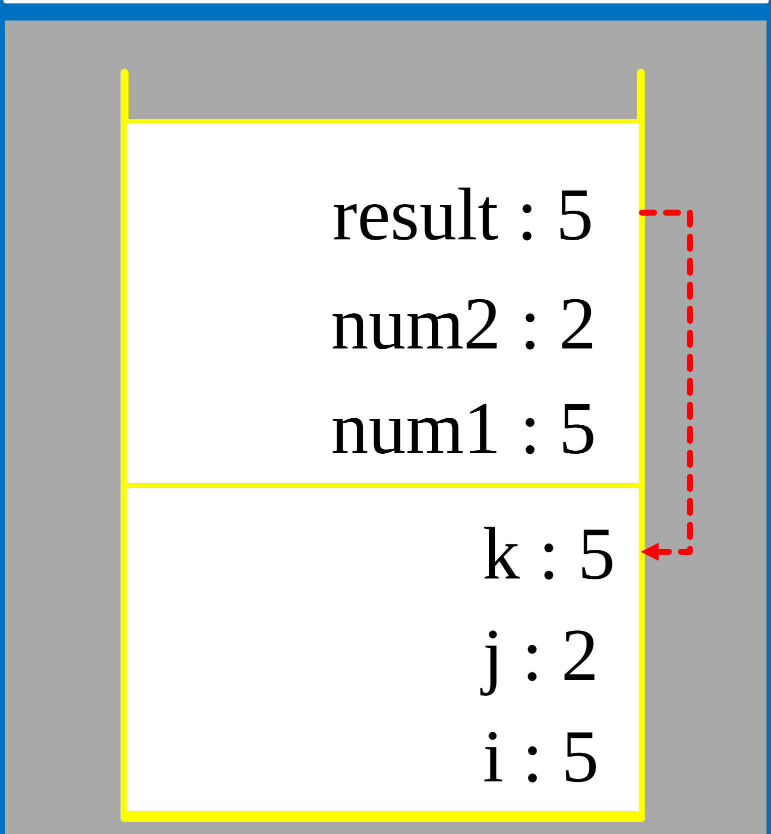
i : 5

The max method is invoked.

Call Stack

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

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

result is returned and assign it to k

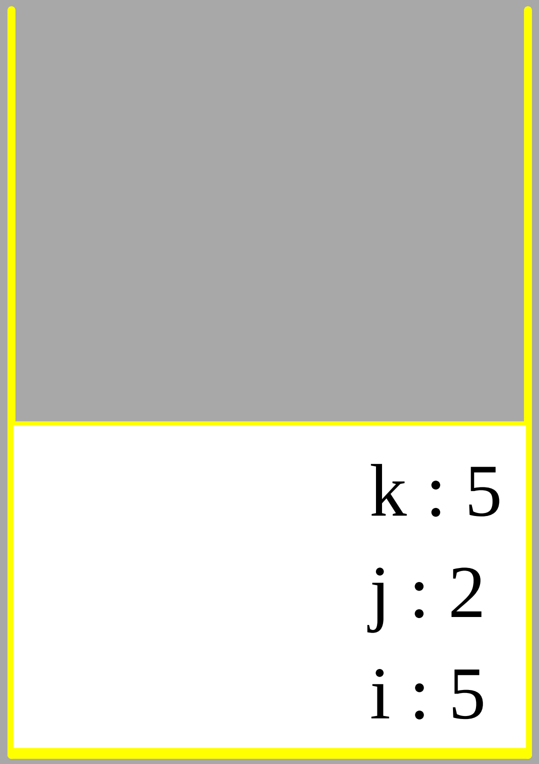


The max method is invoked.

Call Stack

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

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

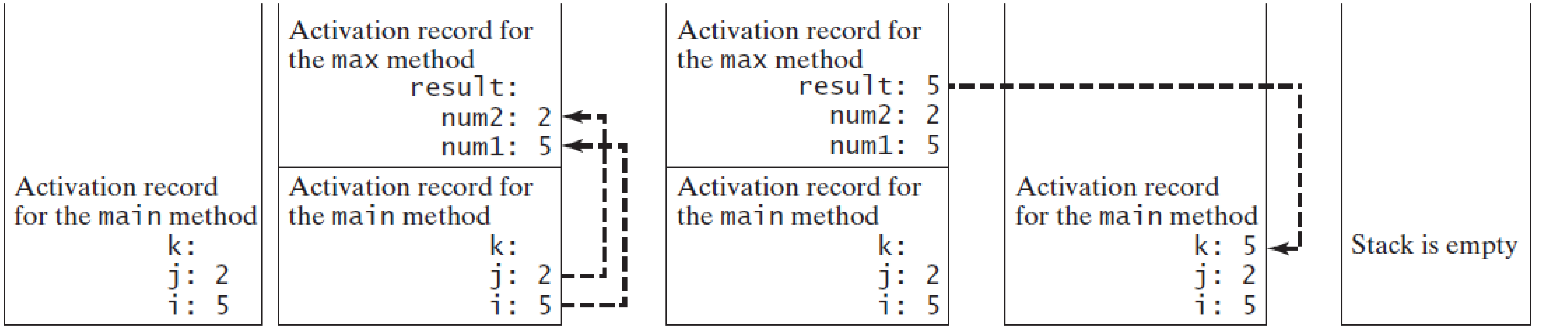
execute print statement



The main method is invoked.

Call Stack

When the *max* method is invoked, the flow of *control transfers to* the *max* method. Once the max method is finished, it *returns control back to the caller*.



Activation record for the main method
k:
j: 2
i: 5

Activation record for the max method
result:
num2: 2
num1: 5

Activation record for the main method
k:
j: 2
i: 5

Activation record for the max method
result: 5
num2: 2
num1: 5

Activation record for the main method
k:
j: 2
i: 5

Activation record for the main method
k: 5
j: 2
i: 5

Stack is empty

(a) The main method is invoked.

(b) The max method is invoked.

(c) The max method is being executed.

(d) The max method is finished and the return value is sent to k.

(e) The main method is finished.

Exercise 1

Identify and correct the errors in the following program:

```
public class Test {
    public static method1(int n, m) {
        n += m;
        method2(3.4);
    }

    public static int method2(int n) {
        if (n > 0) return 1;
        else if (n == 0) return 0;
        else if (n < 0) return -1;
    }
}
```

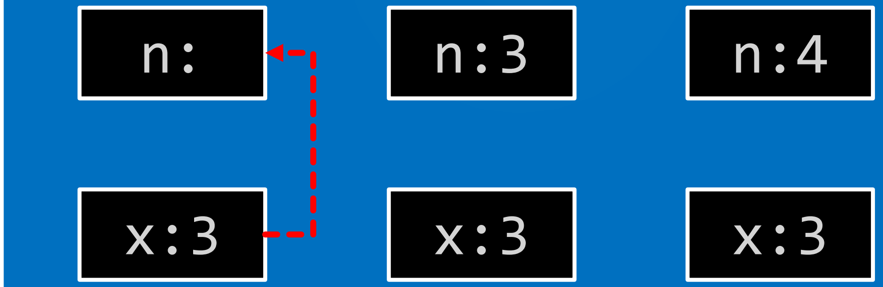
Passing Parameters

The arguments are passed by value to parameters when invoking a method. The arguments must *match* the parameters in *order*, *number*, and *compatible type*, as defined in the method signature. Compatible type means that you can pass an argument to a parameter *without explicit casting*, such as passing an *int* value argument to a *double* value parameter.

Passing Parameters

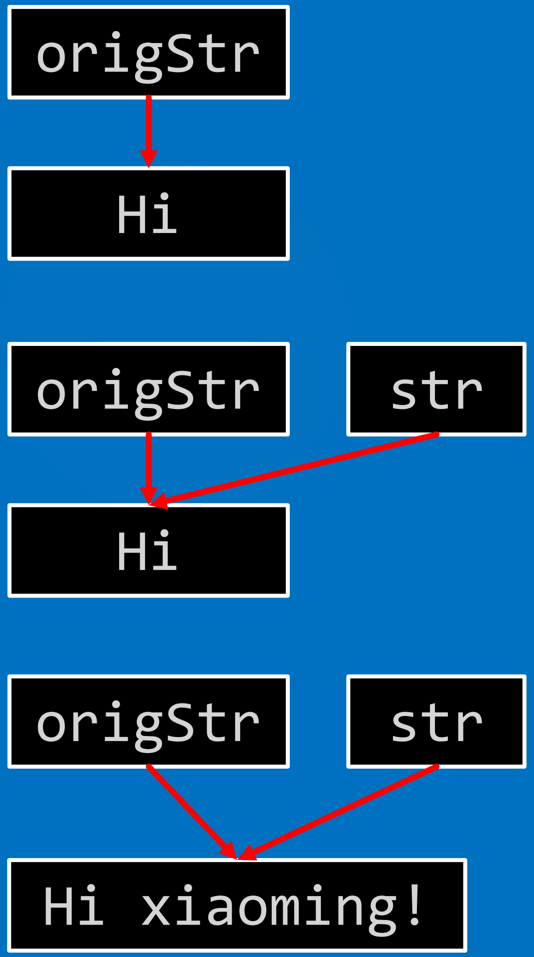
When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. *The variable is not affected*.

```
public class Test {
    public static void main(String[] args) {
        int x = 3;
        increment(x);
        System.out.println("x is " + x);
    }
    public static void increment(int n) {
        n++;
    }
}
```



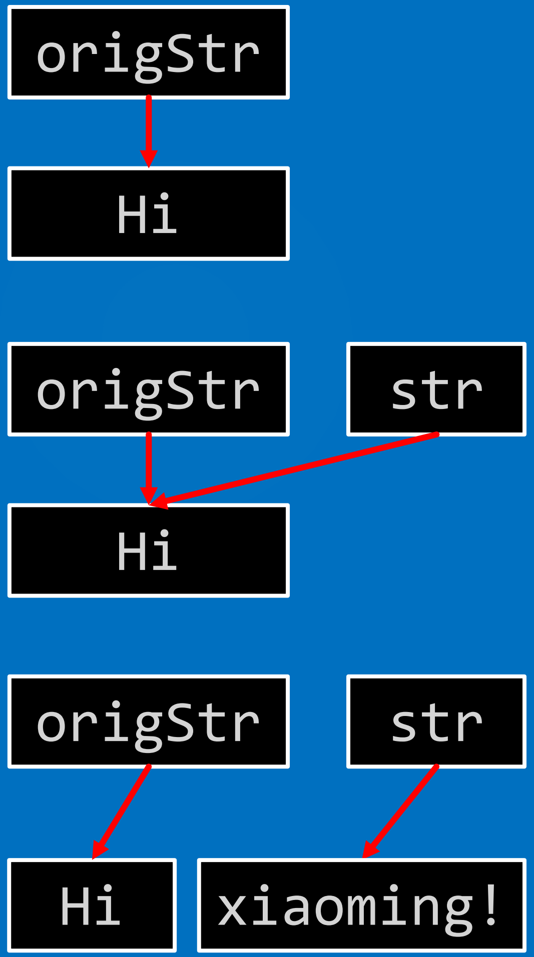
Reference data type parameters

```
public class Test {

    public static void main(String[] args){
        StringBuffer origStr = new StringBuffer("Hi");
        change(origStr);
        System.out.println(origStr);
    }

    public static void change(StringBuffer str){
        str.append(" xiaoming!");
        System.out.println(str);
    }
}
```

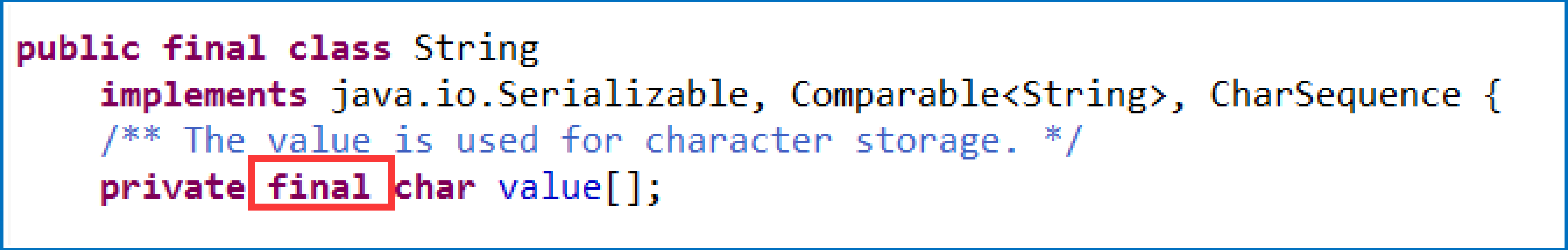


Special Case: *final* modifier in *String*, *Integer*, etc.

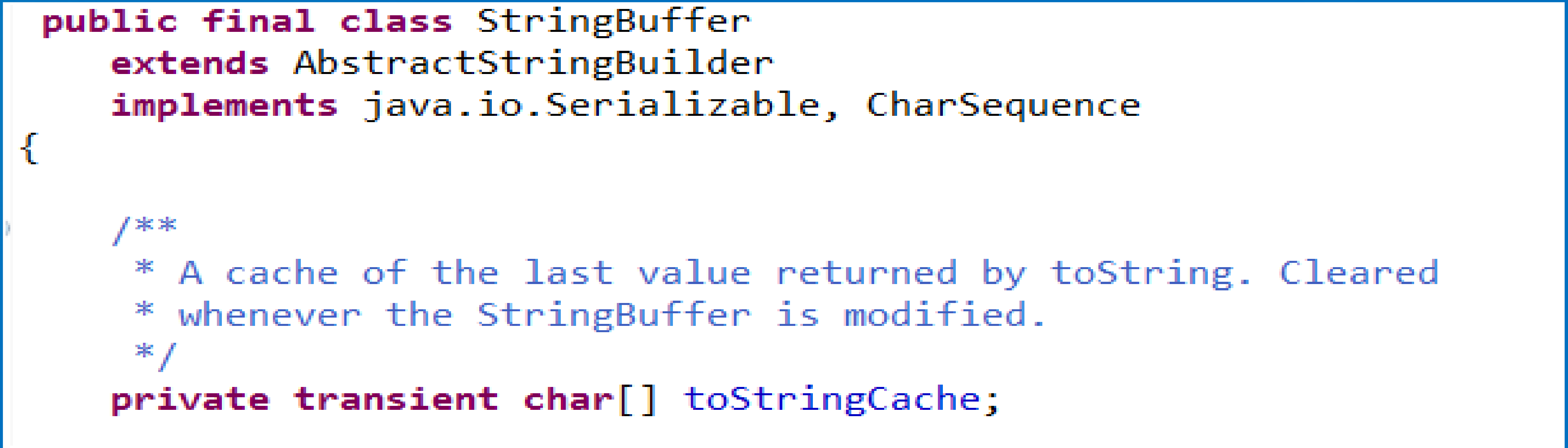
```java
public class Test {
    public static void main(String[] args){
        String origStr = "Hi";
        change(origStr);
        System.out.println(origStr);
    }

    public static void change(String str){
        str += " xiaoming!"; //can't be modified
        System.out.println(str);
    }
}
```



Special Case: *final* modifier in *String*, *Integer*, etc.



```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
```



```
 public final class StringBuffer
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{

    /**
     * A cache of the last value returned by toString. Cleared
     * whenever the StringBuffer is modified.
     */
    private transient char[] toStringCache;
```

Exercise 2

What is *pass-by-value*? Show the result of the following programs.

```java
public class Test {
    public static void main(String[] args) {
        int max = 0;
        max(1, 2, max);
        System.out.println(max);
    }
    public static void max(int value1, int value2, int max) {
        if (value1 > value2)
            max = value1;
        else
            max = value2;
    }
}
```

Modularizing Code

used to *Modularizing* makes the code easy to *maintain* and *debug* and enables the code to be *reused*. Methods can be used to *reduce redundant code* and enable code reuse. Methods can also be modularize code and *improve* the *quality* of the program.

Example: Develop a program that prompts the user to enter *two integers* and displays their *greatest common divisor*. The program with *gcd* method has teveral advantages:

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

GreatestCommonDivisor

GreatestCommonDivisorMethod

Overloading Methods

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

```
public static int max(int a, int b){
   return a > b ? a:b;
}

public static double max(double a, double b){
   return a > b ? a:b;
}
```

Overloading Methods

The Java *Compiler* finds the *most specific method* for a method invocation.

```java
public static void main(String[] args) {
    System.out.println(max(3,5.0));
    System.out.println(max(3.0,5.0));
}
public static double max(double a, double b){
    System.out.println("double and double");
    return a > b?a:b;
}
public static double max(int a, double b){
    System.out.println("int and double");
    return a > b?a:b;
}
```

Overloading Methods: 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 compilation error.

```java
public static void main(String[] args) {
    System.out.println(max(3,5));
}
public static double max(double a, int b){
    System.out.println("double and double");
    return a > b?a:b;
}
public static double max(int a, double b){
    System.out.println("int and double");
    return a > b?a:b;
}
```

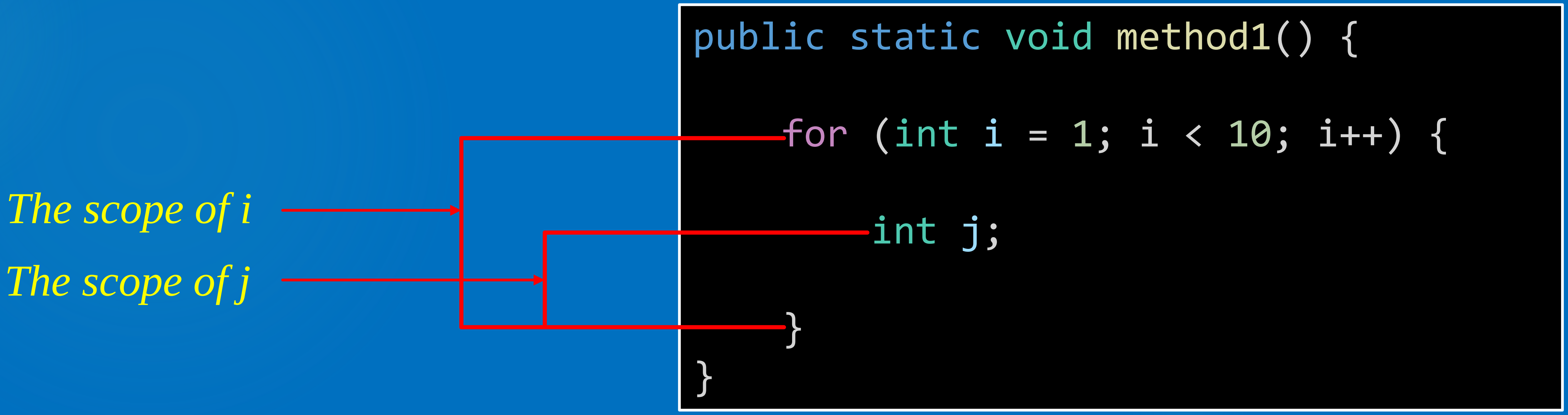
Exercise 3

What is wrong in the following program?

```java
public class Test {
    public static void method(int x) {
    }
    public static int method(int y) {
        return y;
    }
}
```

Scope of Local Variables

The scope of a *local variable* (a variable defined *inside a method*) is 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 and initialized *before* it can be *used*.



```
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

Declare multiple times: 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*.

```
public class Test{
    public static void main(String[] args) {
        for (int i = 1; i < 10; i++) {
            int j;
        }
        int i;
    }
}
```

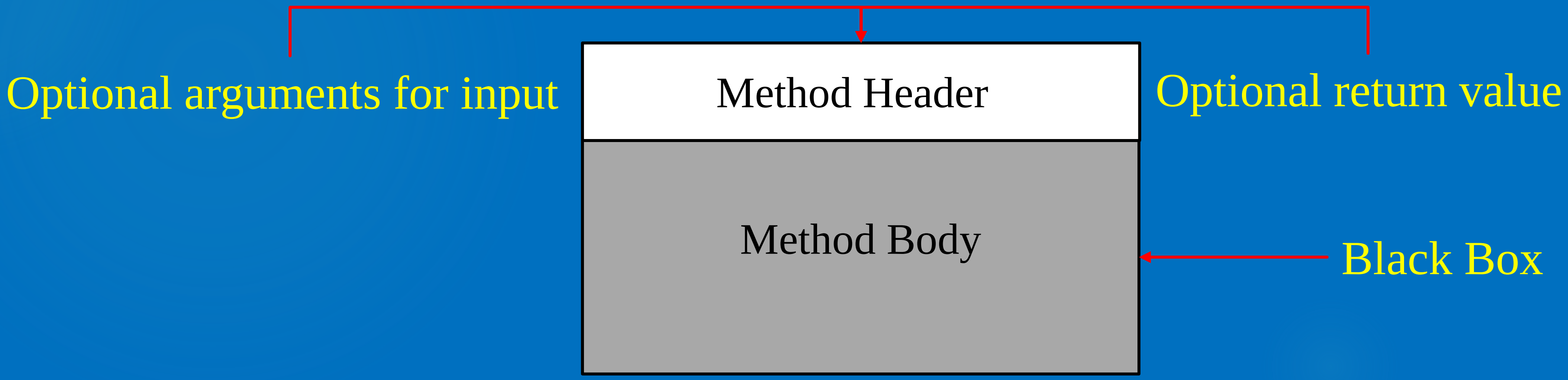
Exercise 4

Identify and correct the errors in the following program:

```java
public class Test {
    public static method1(int n, m) {
        n += m;
        method2(3.4);
    }

    public static int method2(int n) {
        if (n > 0) return 1;
        else if (n == 0) return 0;
        else if (n < 0) return -1;
    }
}
```

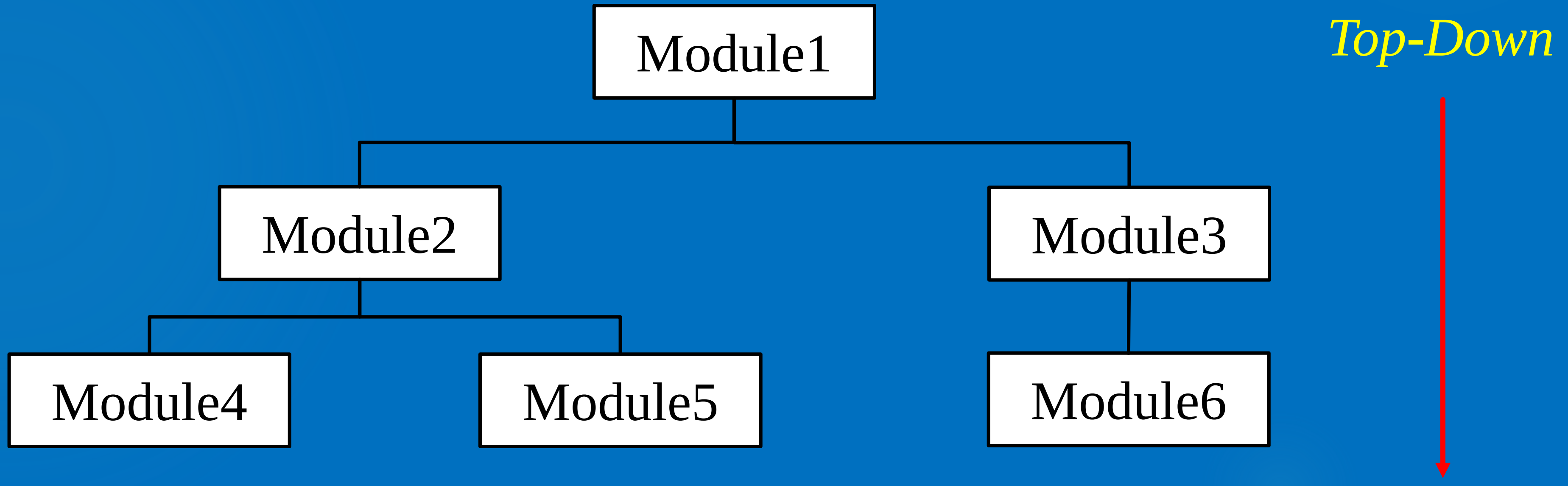
Method Abstraction and Stepwise Refinement

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 who invokes the method. This is also known as *information hiding* or *encapsulation*. If you decide to change the implementation, the client program will not be affected, provided that you do not change the *method signature*. The implementation of the method is hidden from the client in a *“black box”*.



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.



Implementation: *Bottom-Up*

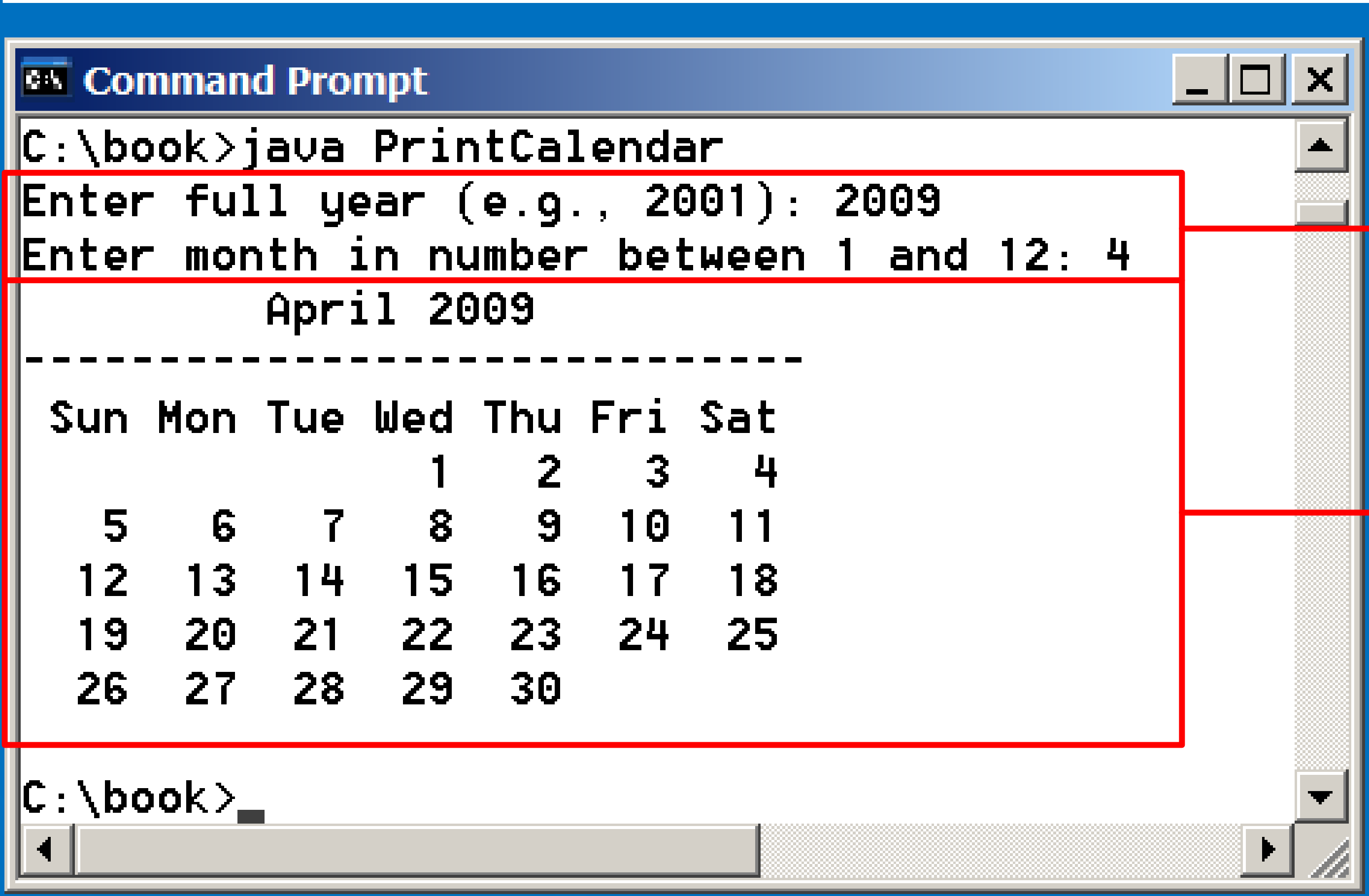
Bottom-up approach is to implement one method in the *structure chart* at a time *from* the *bottom to* the *top*. For each method implemented, write a test program to test it. Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

Stepwise Refinement

Stepwise refinement is the idea that software is developed by *moving through* the levels of abstraction, beginning at *higher levels* and, incrementally *refining* the software through each level of abstraction, *providing more detail at each increment*. *Stepwise refinement breaks* a *large* problem *into smaller* manageable subproblems. Each subproblem can be implemented using a method. This approach makes the program *easier to write, reuse, debug, test, modify, and maintain*.

Case Study: Print a Calender

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

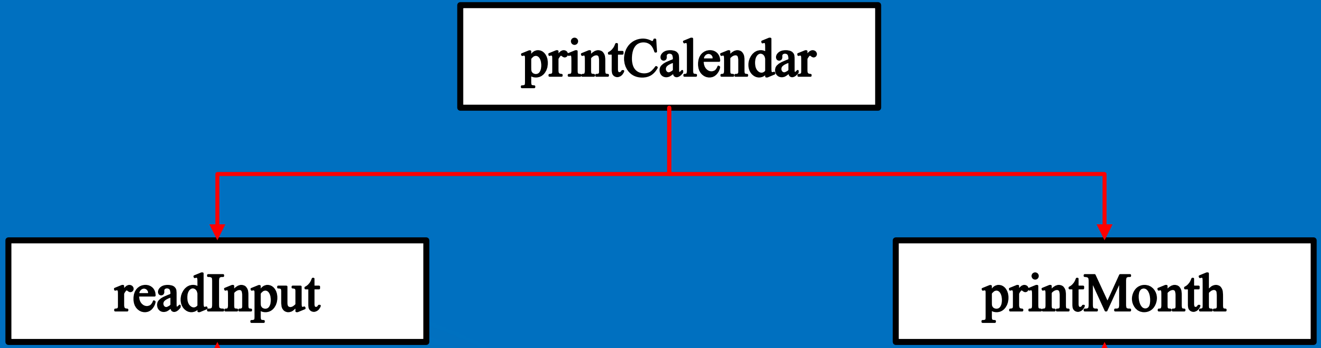


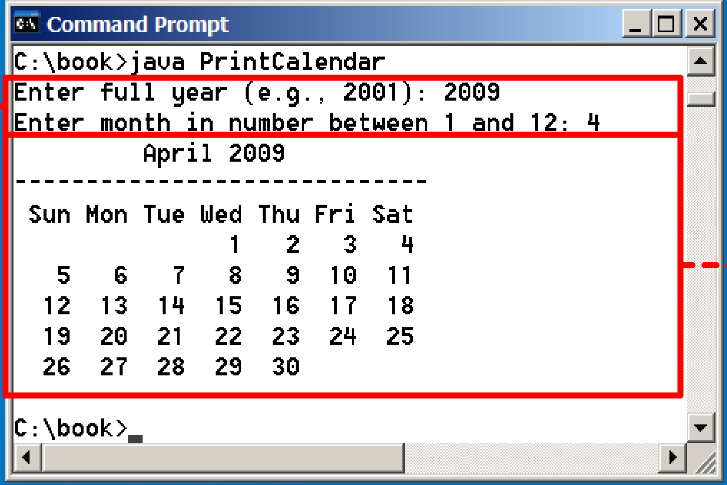
```
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
         April 2009
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
               1   2   3   4
   5   6   7   8   9  10  11
  12  13  14  15  16  17  18
  19  20  21  22  23  24  25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

C:\book>
```

input

printing

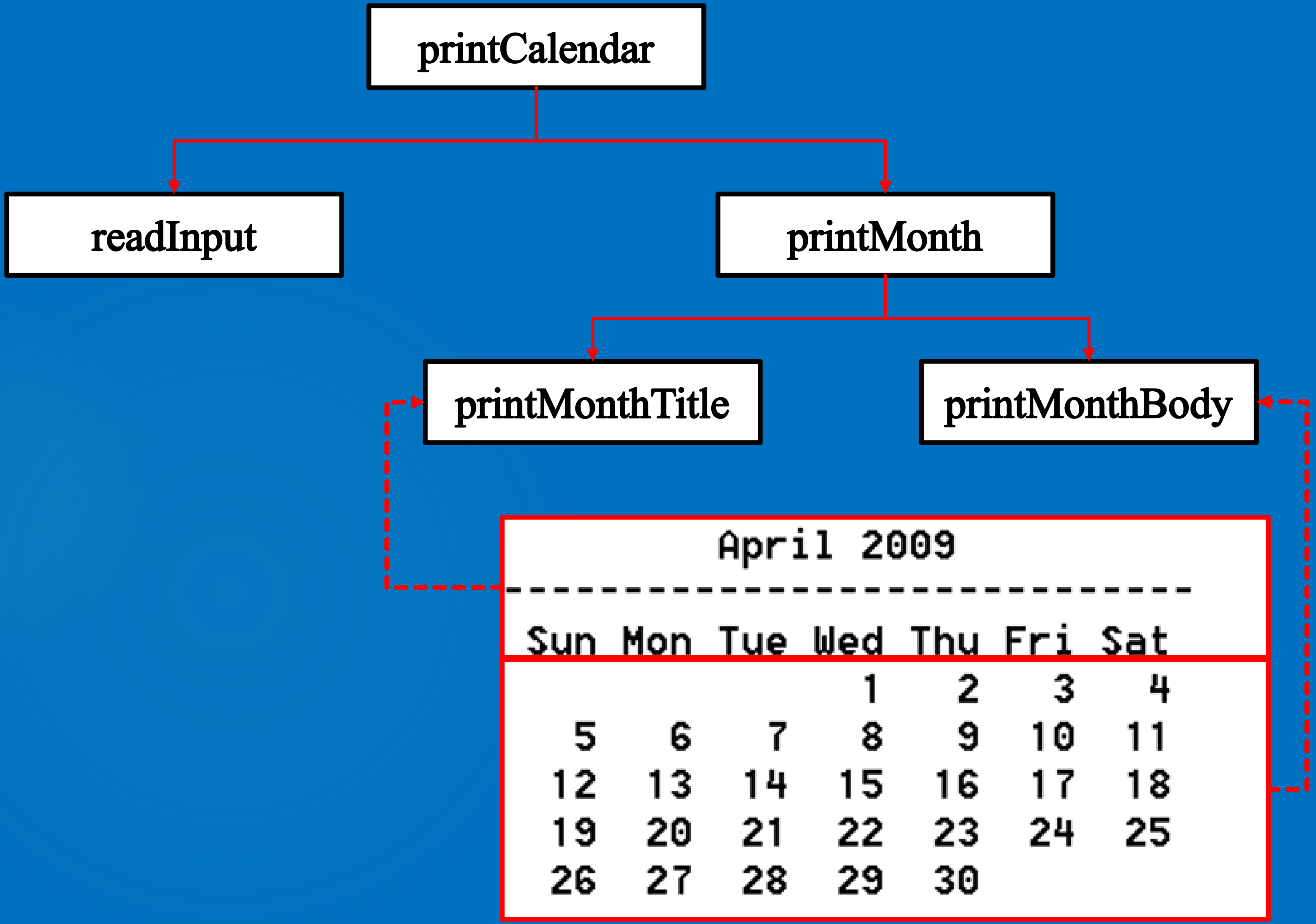
Case Study: Print a Calender





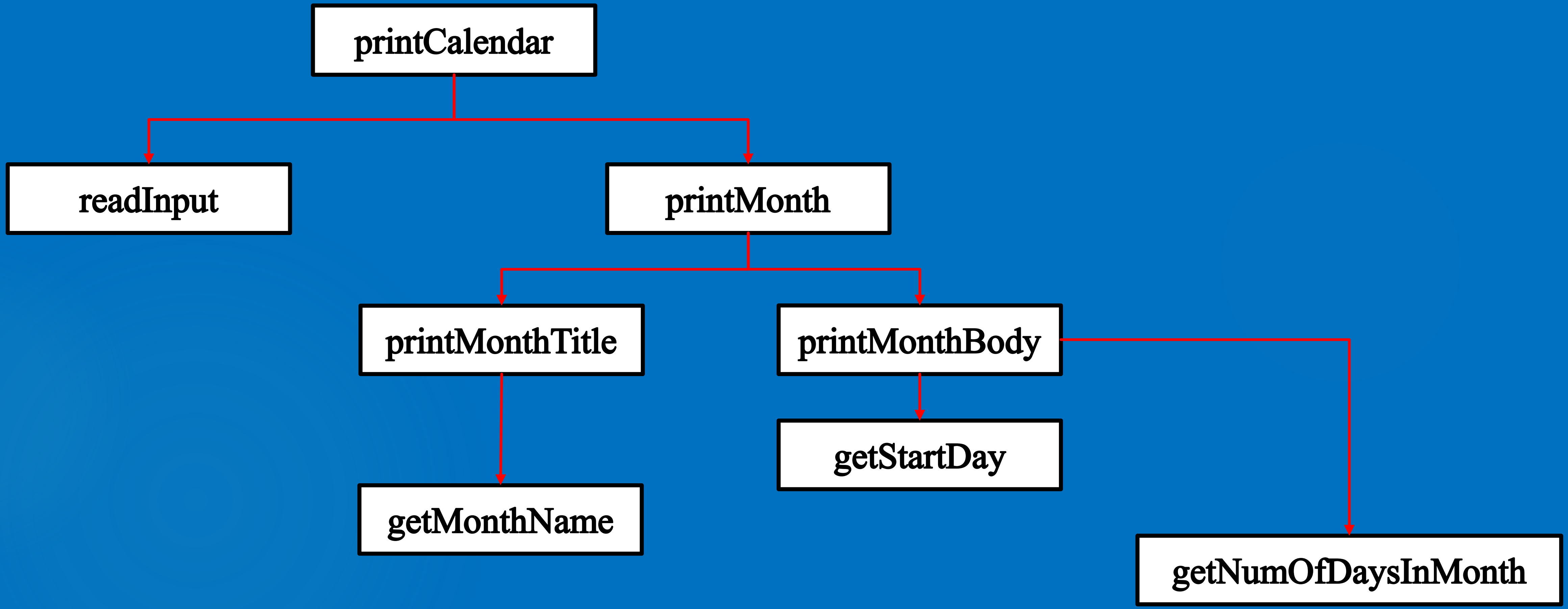
```
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
         April 2009
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
                1   2   3   4
   5   6   7   8   9  10  11
  12  13  14  15  16  17  18
  19  20  21  22  23  24 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

C:\book>_
```

Case Study: Print a Calender

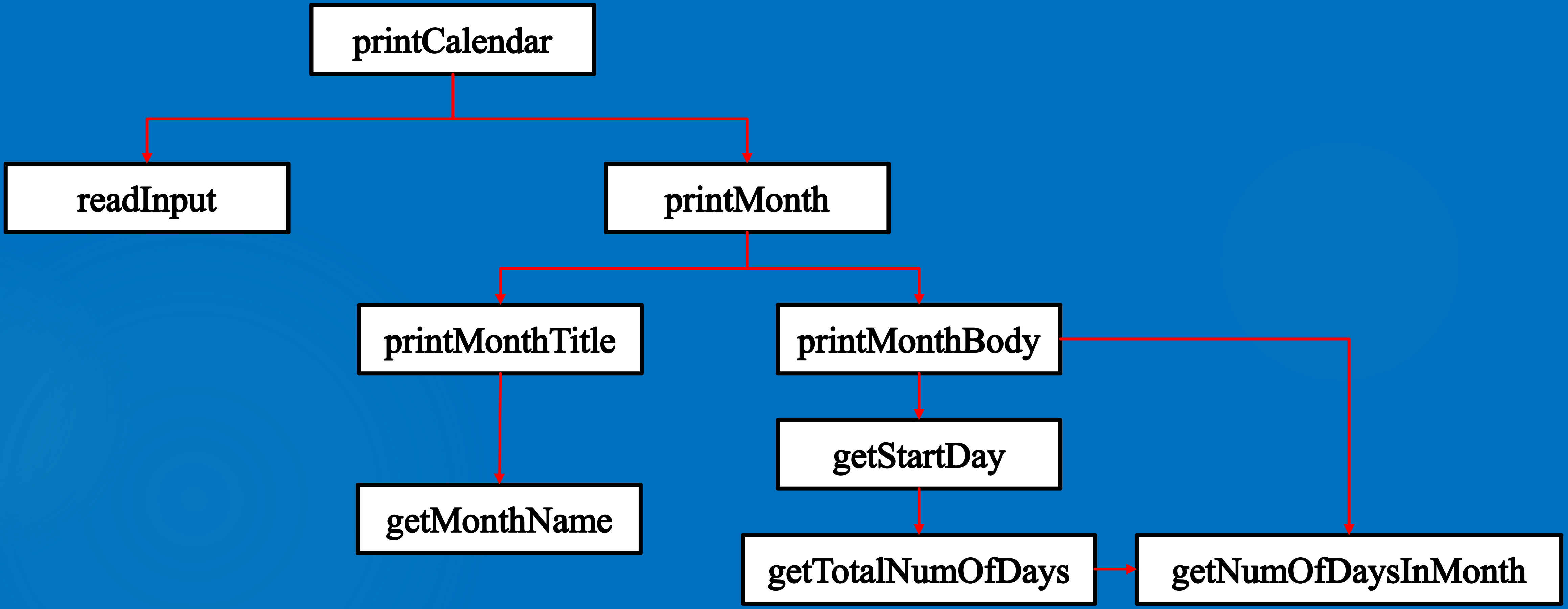


```
         April 2009
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
               1   2   3   4
   5   6   7   8   9  10  11
  12  13  14  15  16  17  18
  19  20  21  22  23  24  25
  26  27  28  29  30
```

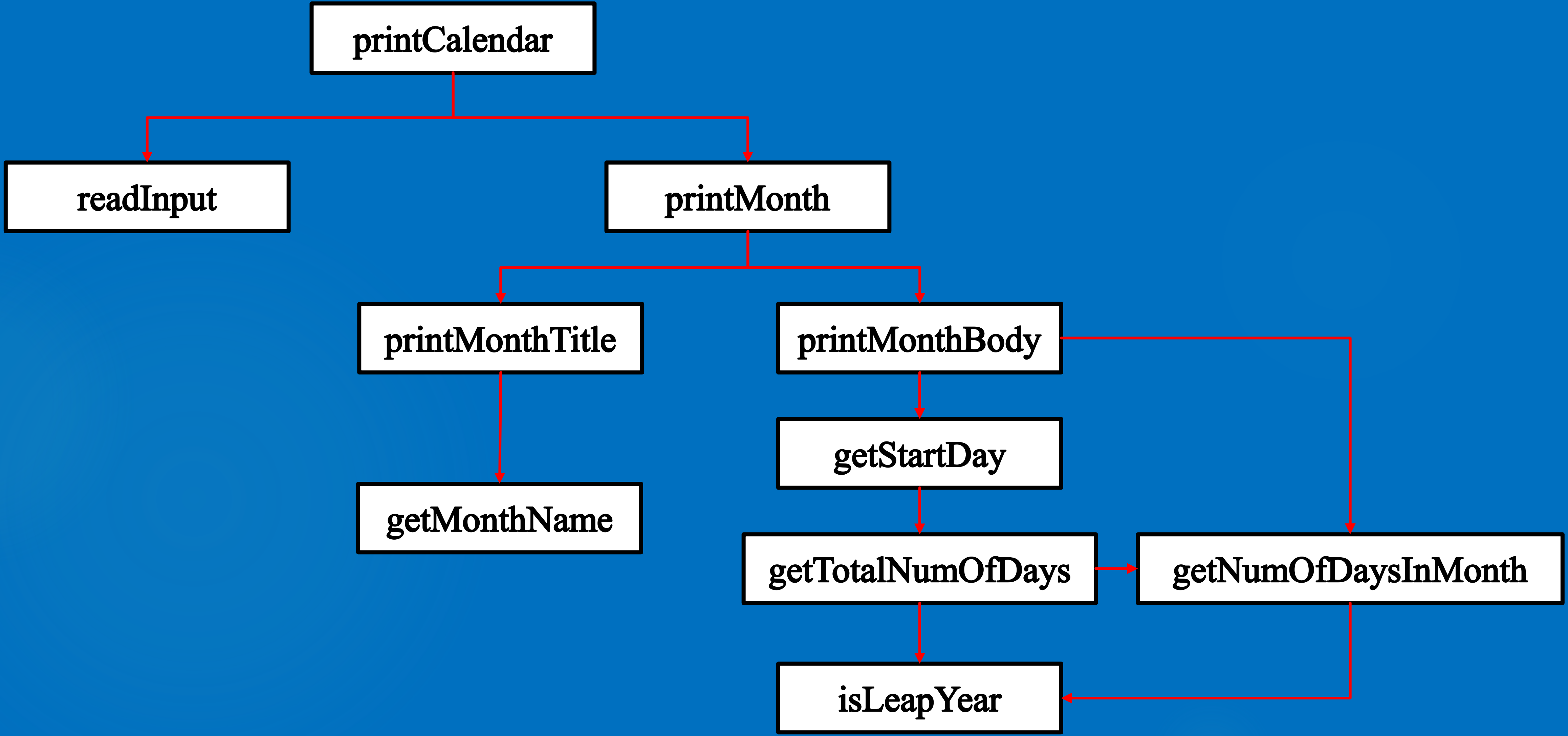
Case Study: Print a Calender



Case Study: Print a Calender



Case Study: Print a Calender


printCalendar
readInput
printMonth
printMonthTitle
printMonthBody
getMonthName
getStartDay
getTotalNumOfDays
getNumOfDaysInMonth
isLeapYear