

Loops and Recursion

Introduction to Programming and
Computational Problem Solving:
Accelerated Pace

CSE 11

Lecture 7

Announcements

- Assignment 3 is due Oct 22, 11:59 PM
 - Upgrade beginning Oct 25, 12:01 AM
- Assignment 4 will be released Oct 22
 - Due Oct 29, 11:59 PM

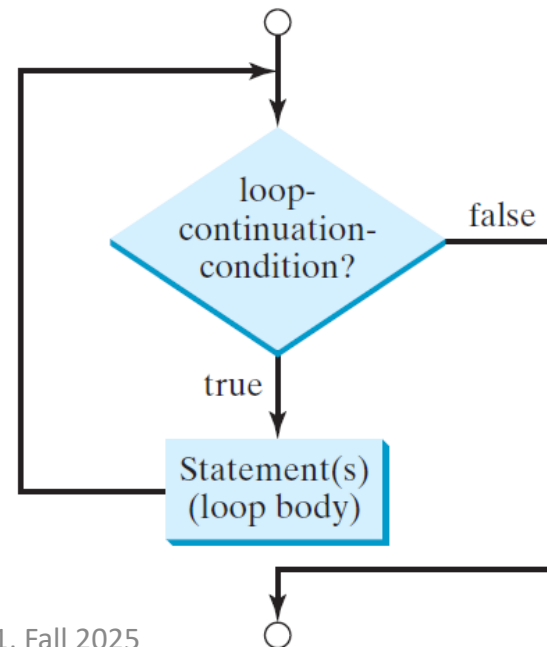
Loops and recursion

- `while` loops
- `do-while` loops
- `for` loops
- Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops
 - A recursive method is one that invokes itself directly or indirectly

while loops

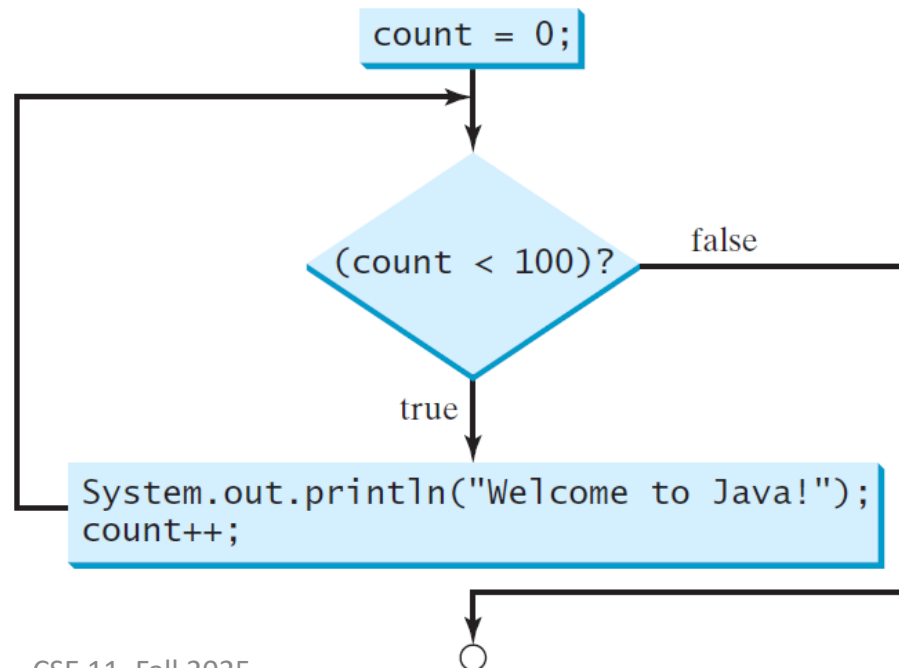
- Executes statements repeatedly while the condition is true

```
while (loop-continuation-condition) {  
    // loop-body  
    Statement(s);  
}
```



while loops

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java");
    count++;
}
```



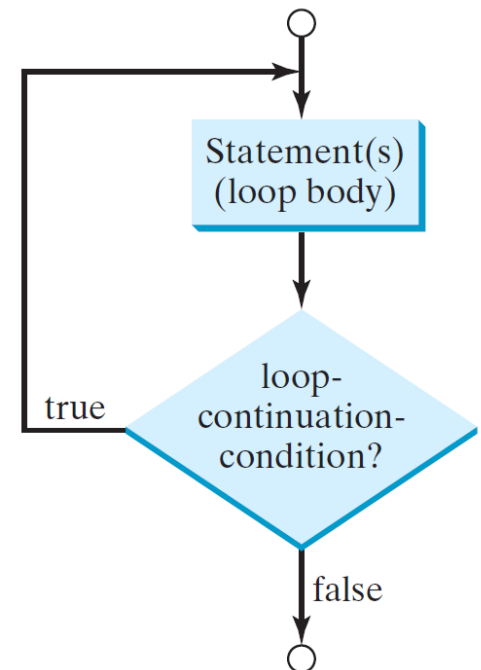
Ending a loop with a sentinel value

- Often the number of times a loop is executed is not predetermined
- You may use an input value to signify the end of the loop
- Such a value is known as a *sentinel value*
- For example, a program reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

do-while loops

- Execute the loop body first, then checks the loop continuation condition

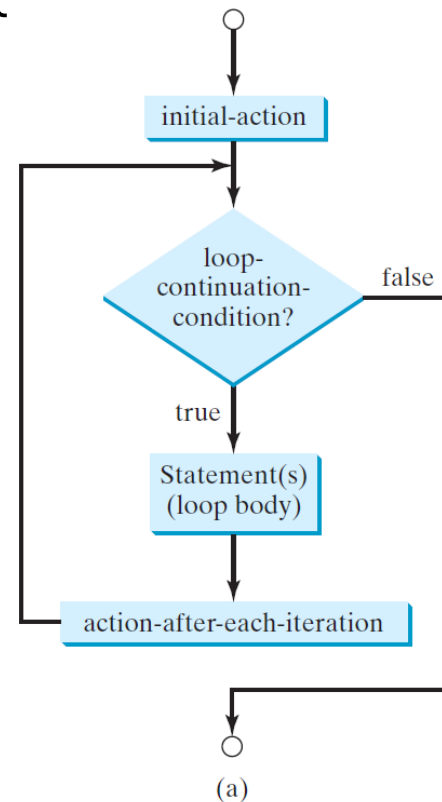
```
do {  
    // Loop body  
    Statement(s);  
} while (loop-continuation-condition);
```



for loops

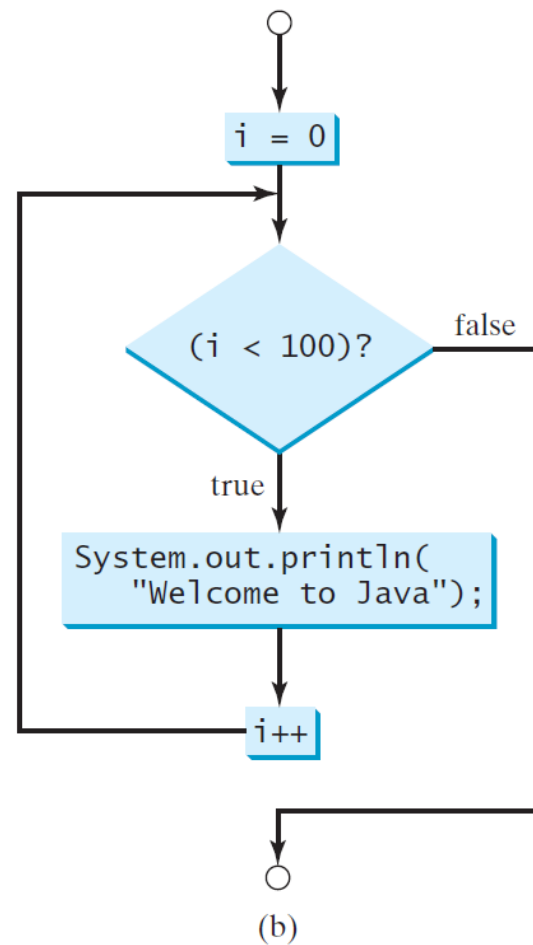
- A concise syntax for writing loops

```
for (initial-action; loop-continuation-condition;  
    action-after-each-iteration) {  
    // loop body  
    Statement(s);  
}
```



for loops

```
int i;  
for (i = 0; i < 100; i++) {  
    System.out.println(  
        "Welcome to Java!");  
}
```



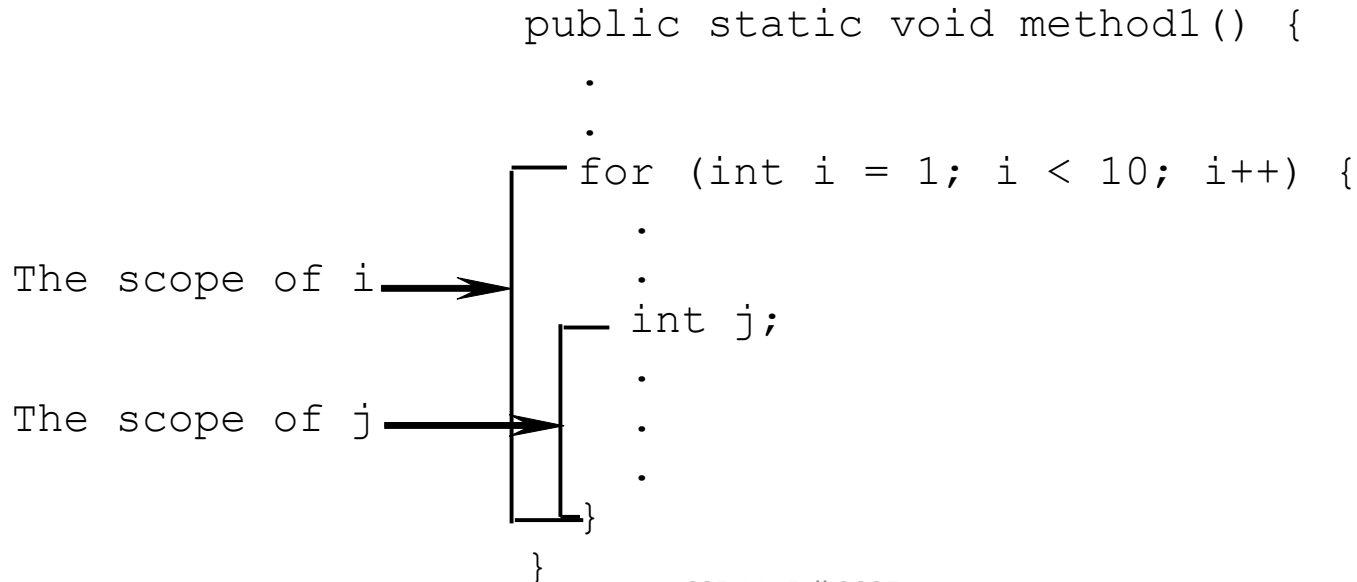
for loops

- The initial-action in a for loop can be a list of zero or more comma-separated expressions
- The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements
- However, it is best practice (less error prone) **not to use comma-separated** expressions and statements

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {  
    // Do something  
}
```

Scope of local variables

- A variable declared in the initial action part of a for loop header has its scope in the entire loop
- A variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable



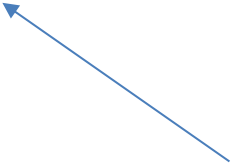
Scope of local variables

```
// 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

```
// With errors
public static void incorrectMethod() {
    int x = 1; // x is declared
    int y = 1;
    for (int i = 1; i < 10; i++) {
        int x = 0;
        x += i;
    }
}
```

Compile error: duplicate local variable



Loops and floating-point accuracy

- Remember, calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy
- As such, **do not use floating-point values for equality checking in a loop control**

```
double sum = 0;
double item = 1;
while (item != 0) { // No guarantee item will be 0
    sum += item;
    item -= 0.1;
}
System.out.println(sum);
```

Infinite loops

- If the loop-continuation-condition in a for loop is omitted, it is implicitly true

```
for ( ; ; ) {  
    // Do something  
}
```

(a)

Equivalent

```
while (true) {  
    // Do something  
}
```

(b)

Loops

- The three forms of loop statements, `while`, `do-while`, and `for`, are expressively equivalent
 - You can write a loop in any of these three forms

```
while (loop-continuation-condition) {  
    // Loop body  
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; )  
    // Loop body  
}
```

(b)

```
for (initial-action;  
     loop-continuation-condition;  
     action-after-each-iteration) {  
    // Loop body;  
}
```

(a)

Equivalent

```
initial-action;  
while (loop-continuation-condition) {  
    // Loop body;  
    action-after-each-iteration;  
}
```

(b)

Loops

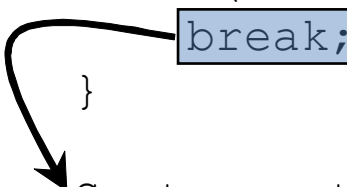
- Use the loop form that is most intuitive and comfortable
 - A `for` loop may be used if the number of repetitions is known
 - A `while` loop may be used if the number of repetitions is not known
 - A `do-while` loop can be used to replace a `while` loop if the loop body must be executed before testing the continuation condition

break

- Immediately terminate the loop

```
public class TestBreak {
    public static void main(String[] args) {
        int sum = 0;
        int number = 0;

        while (number < 20) {
            number++;
            sum += number;
            if (sum >= 100)
                break;
        }
        System.out.println("The number is " + number);
        System.out.println("The sum is " + sum);
    }
}
```



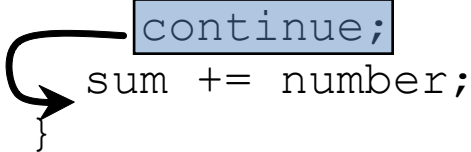
continue

- End the current iteration
 - Program control goes to the end of the loop body

```
public class TestContinue {
    public static void main(String[] args) {
        int sum = 0;
        int number = 0;

        while (number < 20) {
            number++;
            if (number == 10 || number == 11)
                continue;
            sum += number;
        }

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



Nested loops

- Loops can be nested
- For example, nested for loops are often used to handle two-dimensional data

```
for (int i = 0; i < numRows; i++) {  
    // Handle i-th row  
    for (int j = 0; j < numColumns; j++) {  
        // Handle j-th column on i-th row  
    }  
}
```

Recursion

- Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops
- A recursive method is one that invokes itself directly or indirectly

Computing factorials

- Example

$$4! = 4 * 3 * 2 * 1 = 24$$

- Remember, $0! = 1$ (and $1! = 1$)
- As a (non-recursive) method

```
public static long factorial(int n) {  
    long nfactorial = 0 == n ? 1 : n;  
    for (int i = n - 1; 1 < i; --i) {  
        nfactorial *= i;  
    }  
    return nfactorial;  
}
```

Computing factorials

- Alternatively, think recursively

$$0! = 1$$

- *Base case or stopping condition*

$$n! = n * (n - 1)!; n > 0$$

- $(n - 1)!$ is a *subproblem* of $n!$ and is a *recursive call*

- Example

$$4! = 4 * 3!$$

$$4! = 4 * (3 * 2!)$$

$$4! = 4 * (3 * (2 * 1!))$$

$$4! = 4 * (3 * (2 * (1 * 0!)))$$

$$4! = 4 * (3 * (2 * (1 * 1)))$$

$$4! = 4 * (3 * (2 * 1))$$

$$4! = 4 * (3 * 2)$$

$$4! = 4 * 6$$

$$4! = 24$$

Computing factorials

$$0! = 1$$

$$\text{factorial}(0) = 1$$

$$n! = n * (n - 1)!; n > 0$$

$$\text{factorial}(n) = n * \text{factorial}(n - 1)$$

- As a recursive method

```
public static long factorial(int n) {  
    if (0 == n) {  
        // Base case  
        return 1;  
    }  
    else {  
        // Recursive call  
        return n * factorial(n - 1);  
    }  
}
```

Computing factorials

- Example

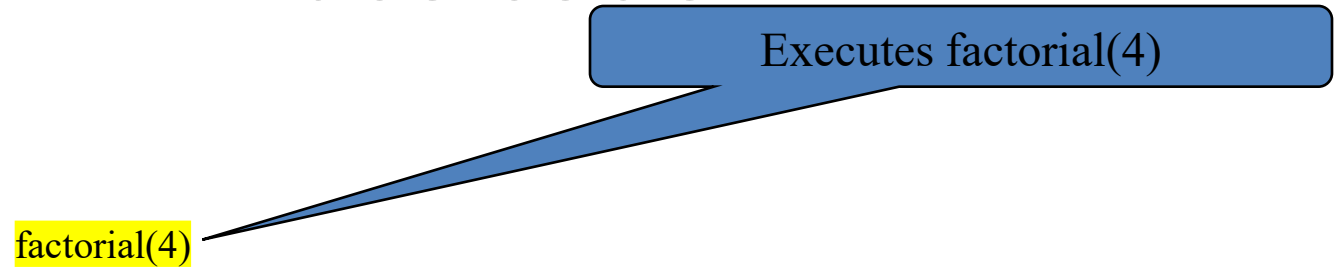
```
0! = 1
n! = n * (n - 1)!; n > 0
```

```
4! = 4 * 3!
4! = 4 * (3 * 2!)
4! = 4 * (3 * (2 * 1!))
4! = 4 * (3 * (2 * (1 * 0!)))
4! = 4 * (3 * (2 * (1 * 1)))
4! = 4 * (3 * (2 * 1))
4! = 4 * (3 * 2)
4! = 4 * 6
4! = 24
```

```
factorial(0) = 1
factorial(n) = n * factorial(n - 1)
```

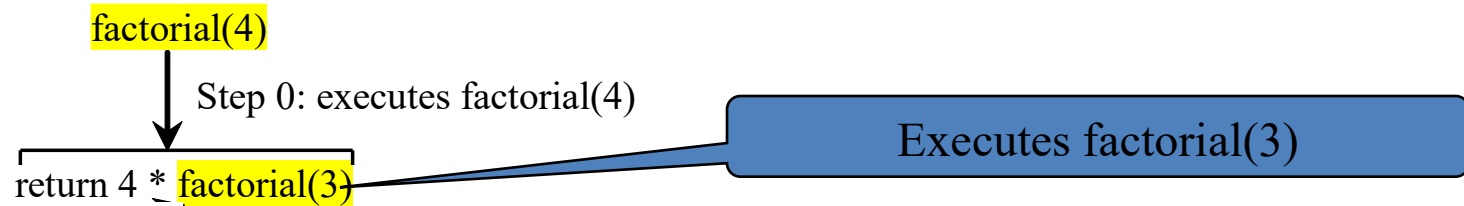
```
factorial(4) = 4 * factorial(3)
factorial(4) = 4 * (3 * factorial(2))
factorial(4) = 4 * (3 * (2 * factorial(1)))
factorial(4) = 4 * (3 * (2 * (1 * factorial(0))))
factorial(4) = 4 * (3 * (2 * (1 * 1)))
factorial(4) = 4 * (3 * (2 * 1))
factorial(4) = 4 * (3 * 2)
factorial(4) = 4 * 6
factorial(4) = 24
```

Trace code



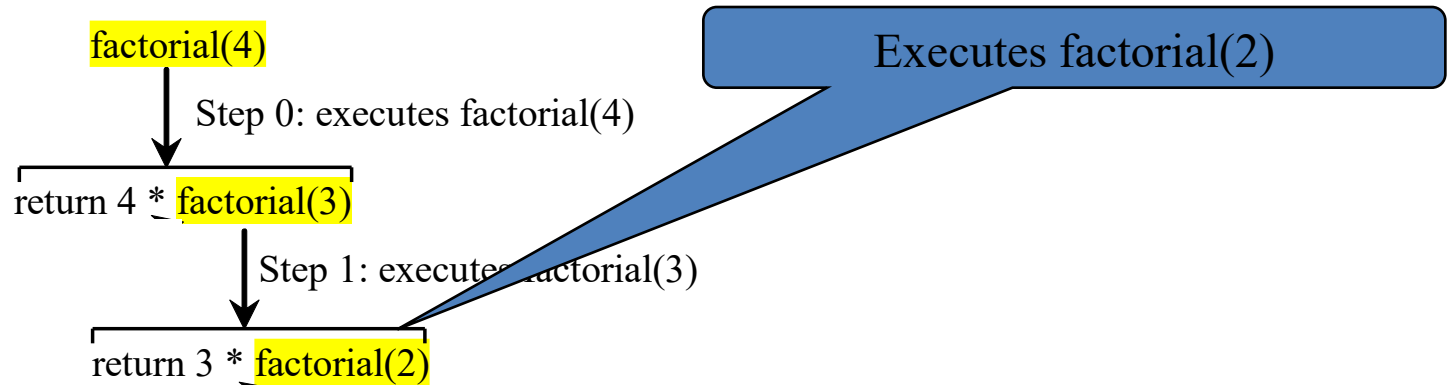
| |
|------------------------------------|
| Stack |
| Space Required for factorial(4) |
| Main method |

Trace code



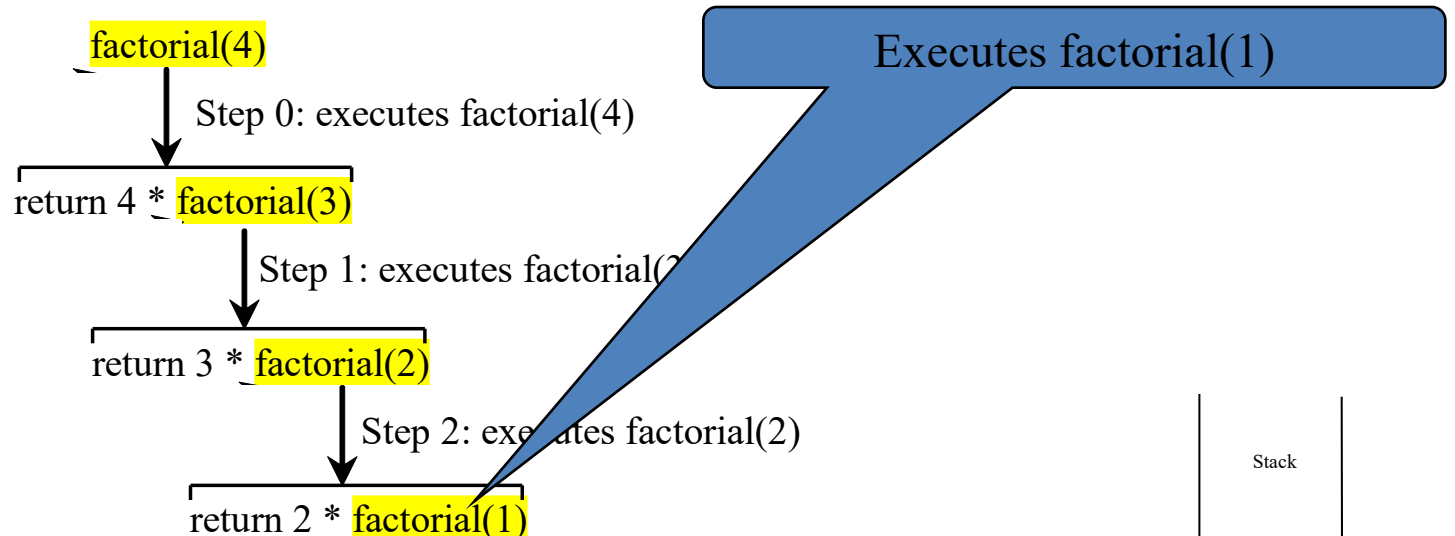
| |
|---------------------------------|
| Stack |
| Space Required for factorial(3) |
| Space Required for factorial(4) |
| Main method |

Trace code



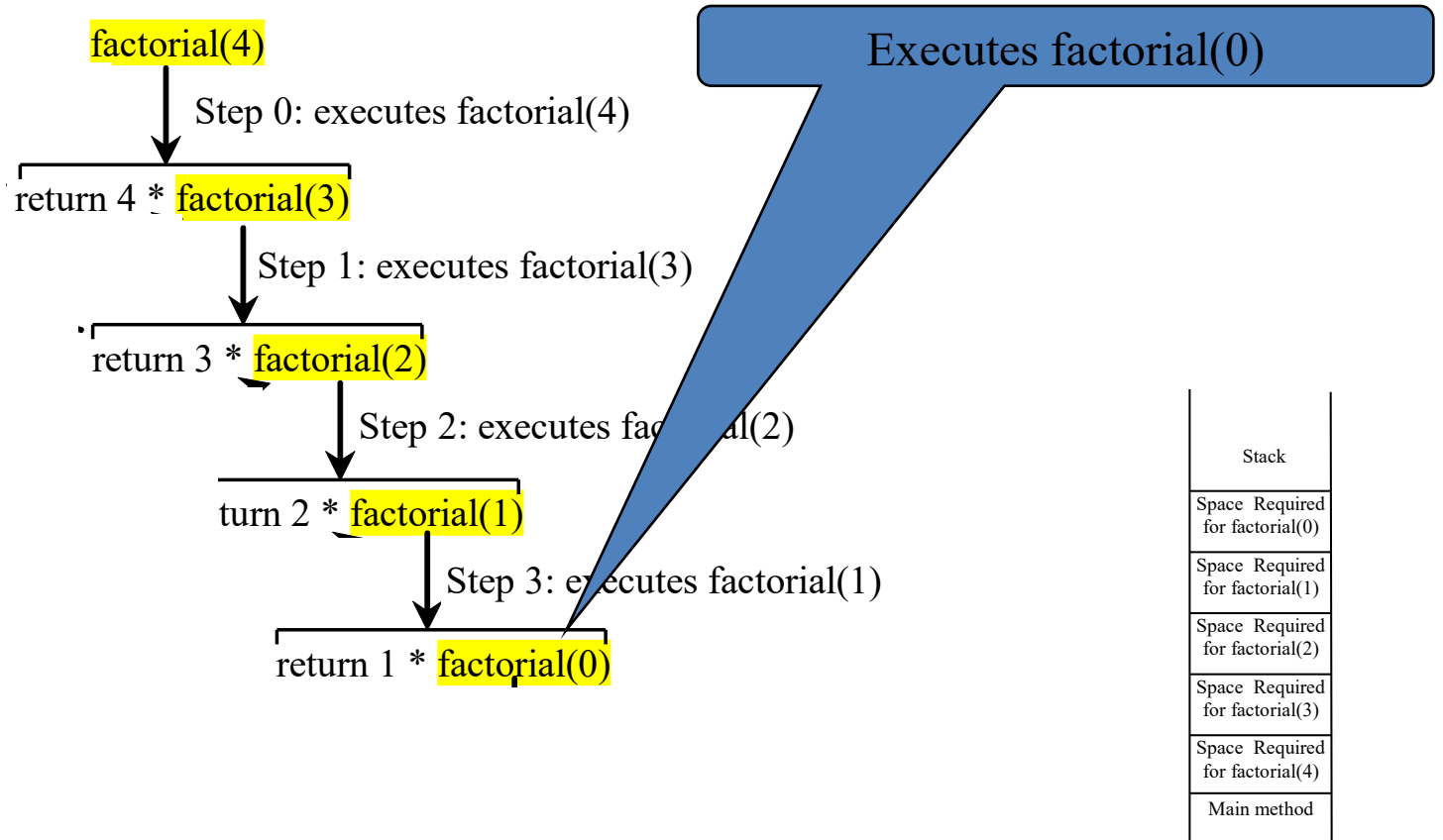
| |
|---------------------------------|
| Stack |
| Space Required for factorial(2) |
| Space Required for factorial(3) |
| Space Required for factorial(4) |
| Main method |

Trace code

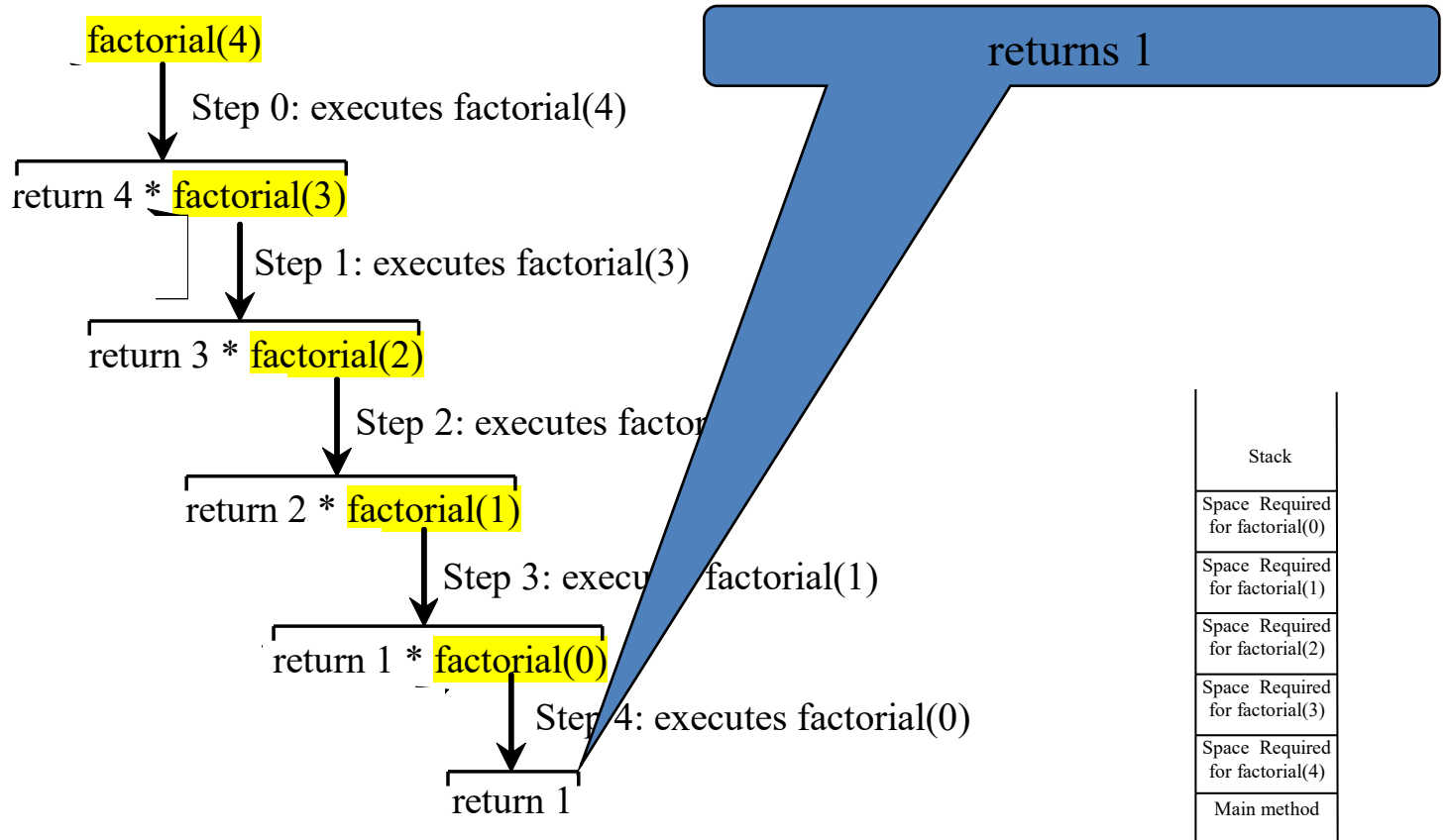


| |
|---------------------------------|
| Stack |
| Space Required for factorial(1) |
| Space Required for factorial(2) |
| Space Required for factorial(3) |
| Space Required for factorial(4) |
| Main method |

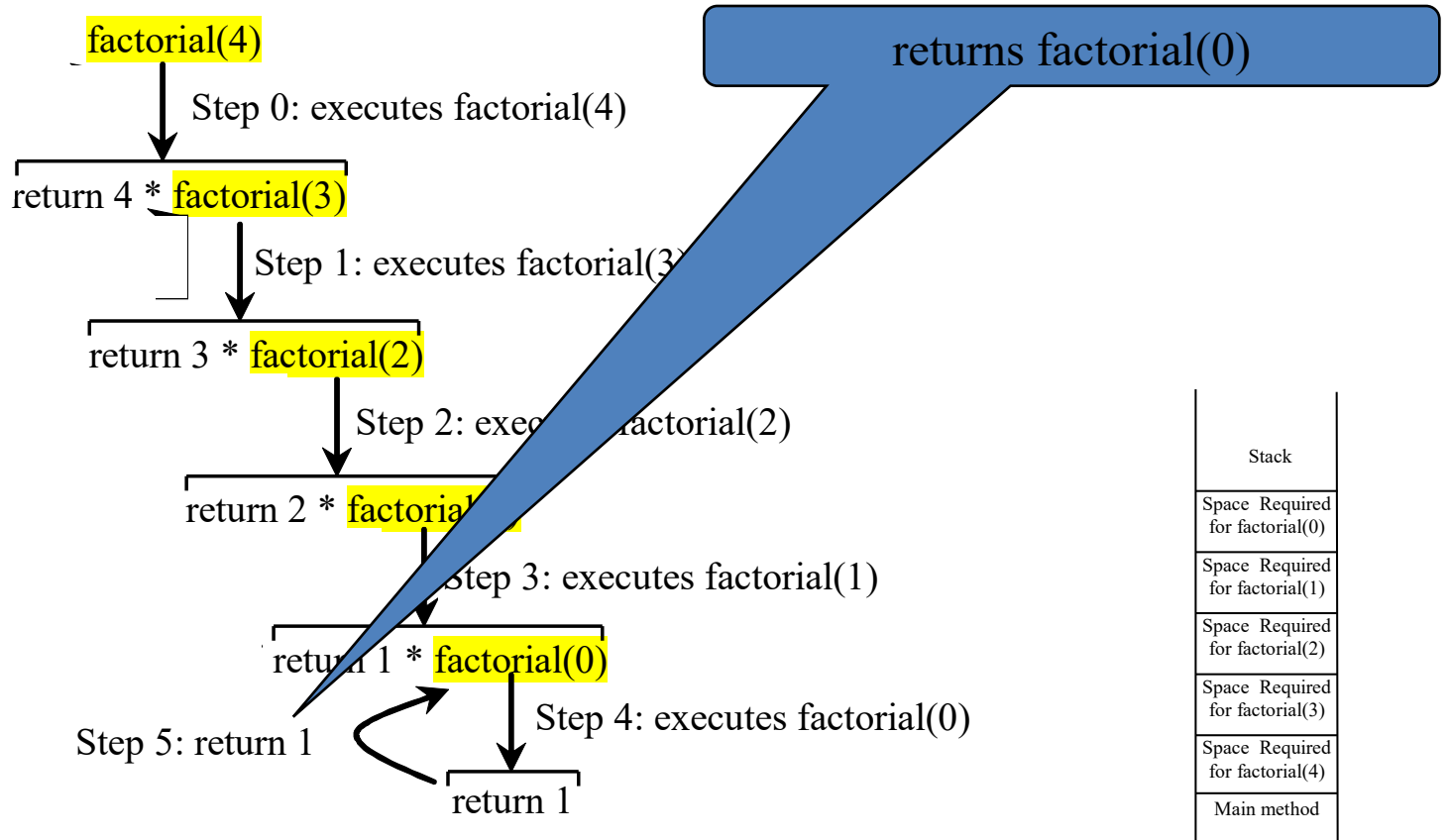
Trace code



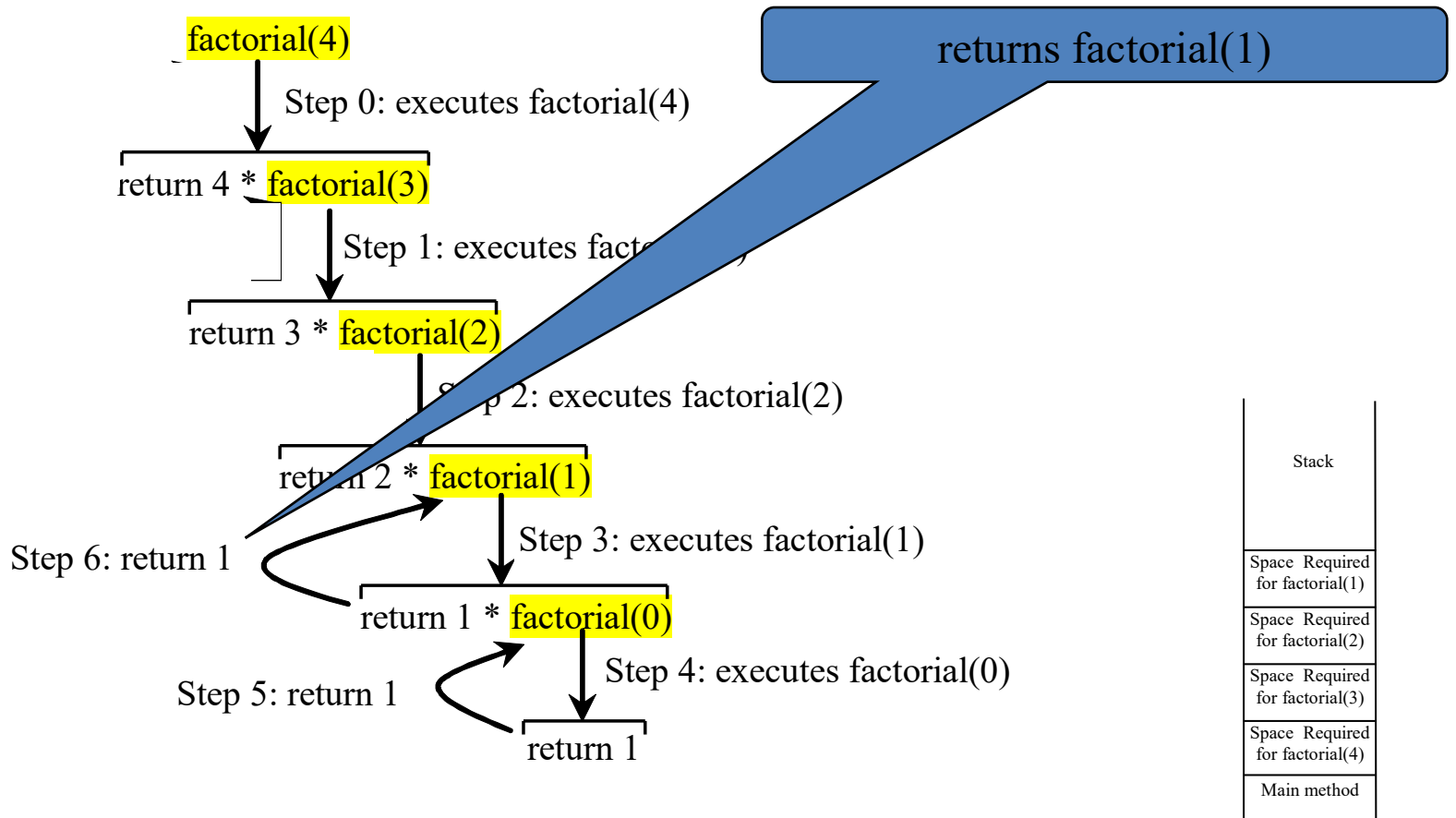
Trace code



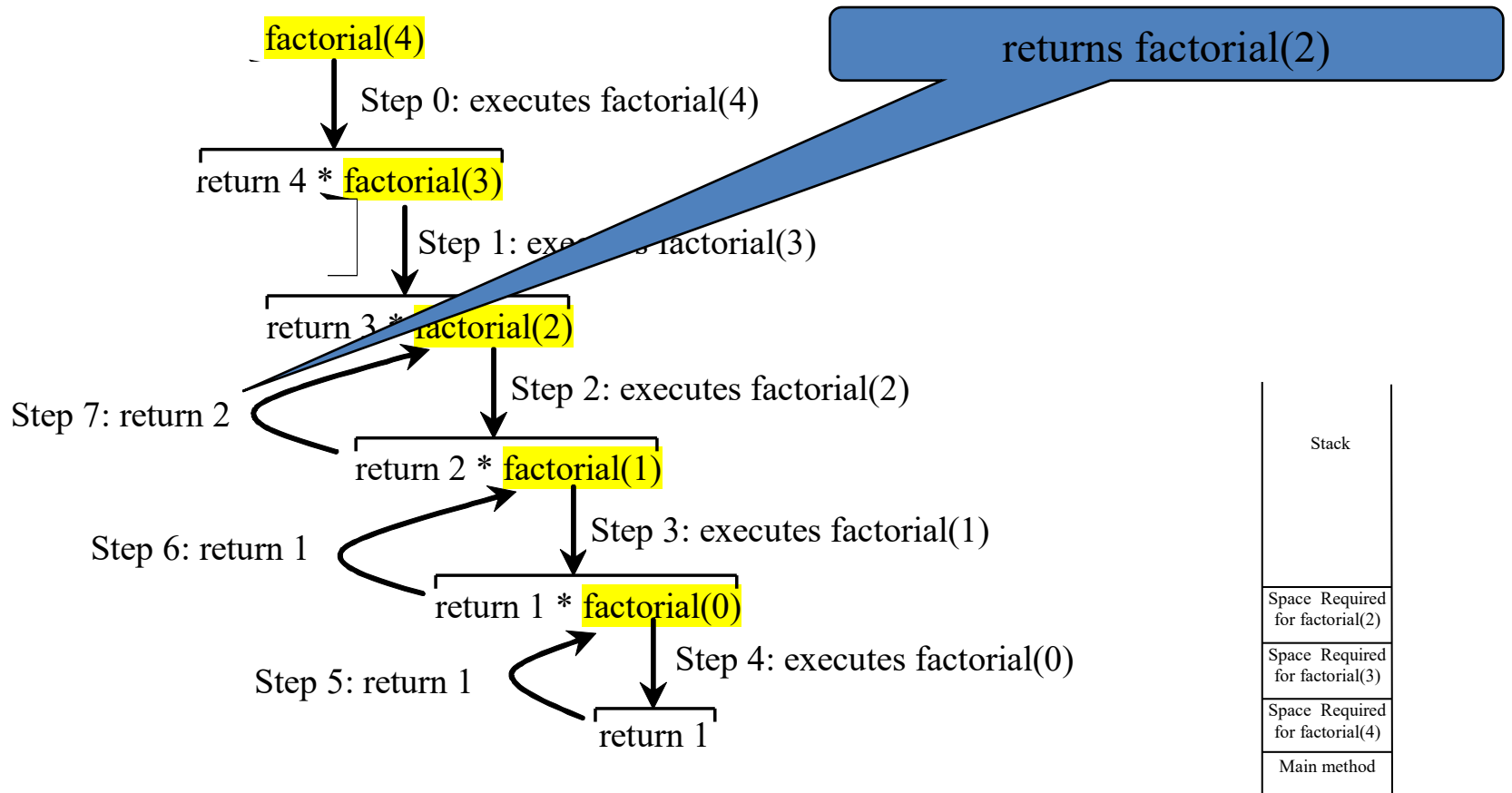
Trace code



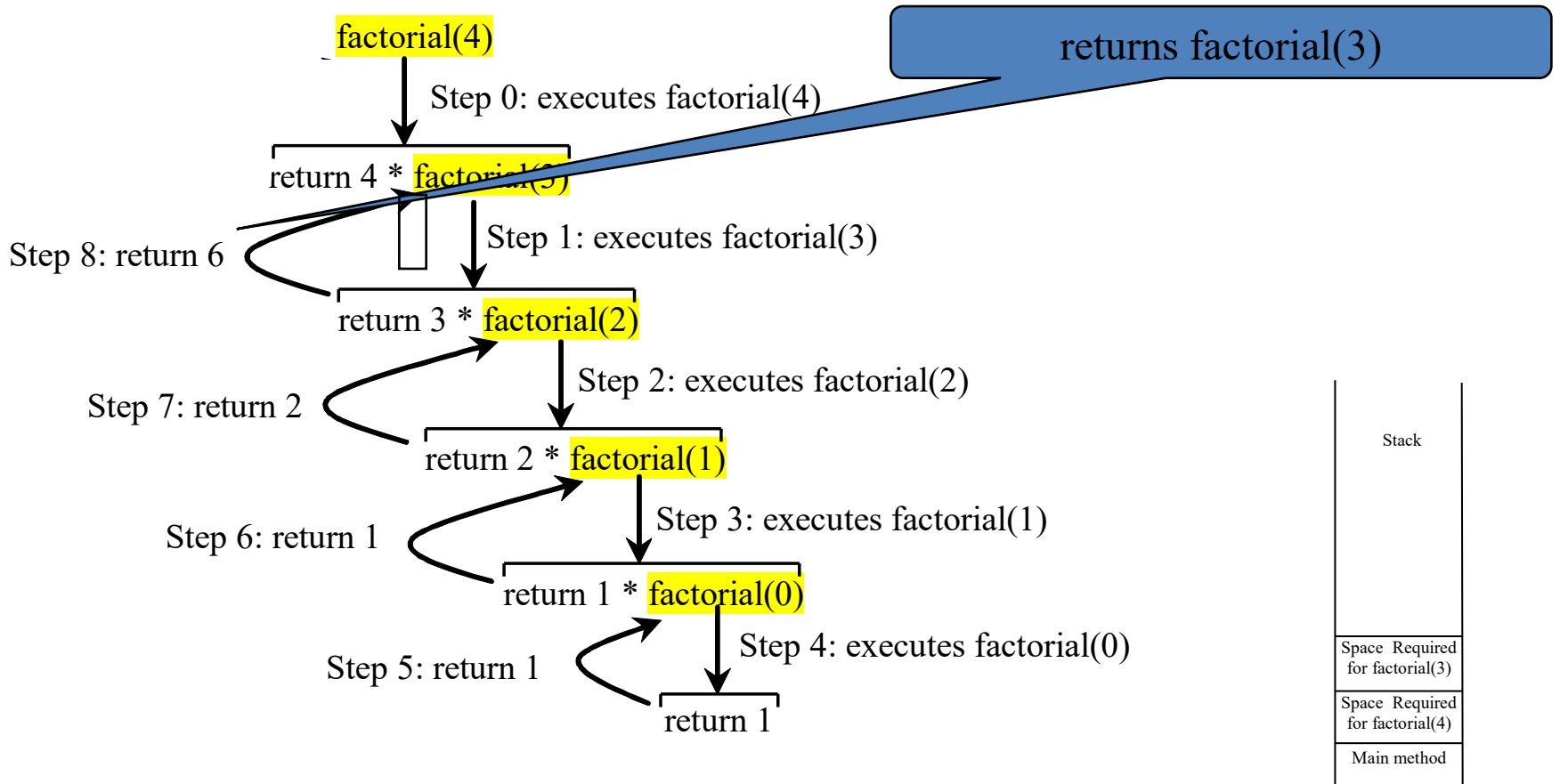
Trace code



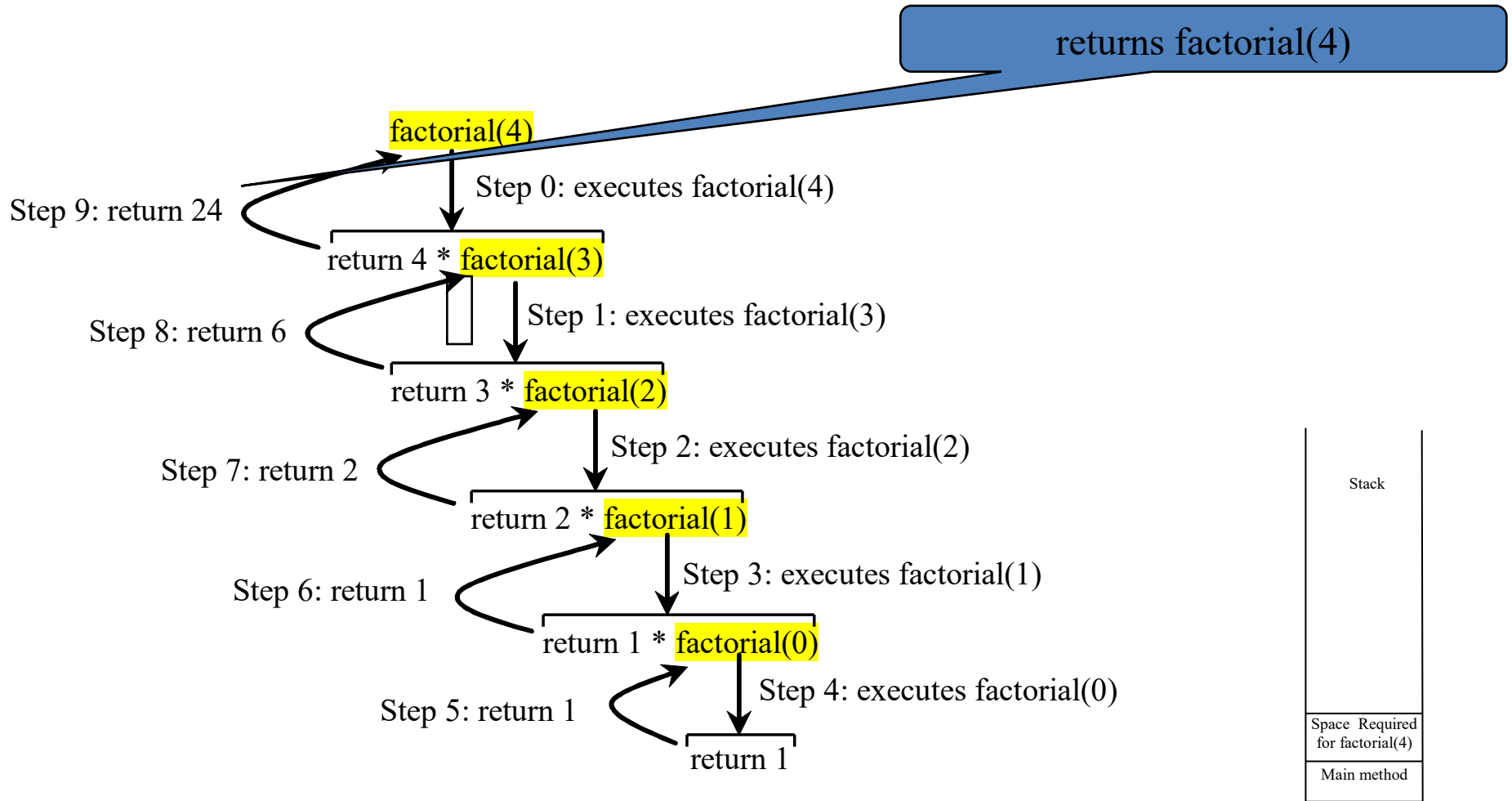
Trace code



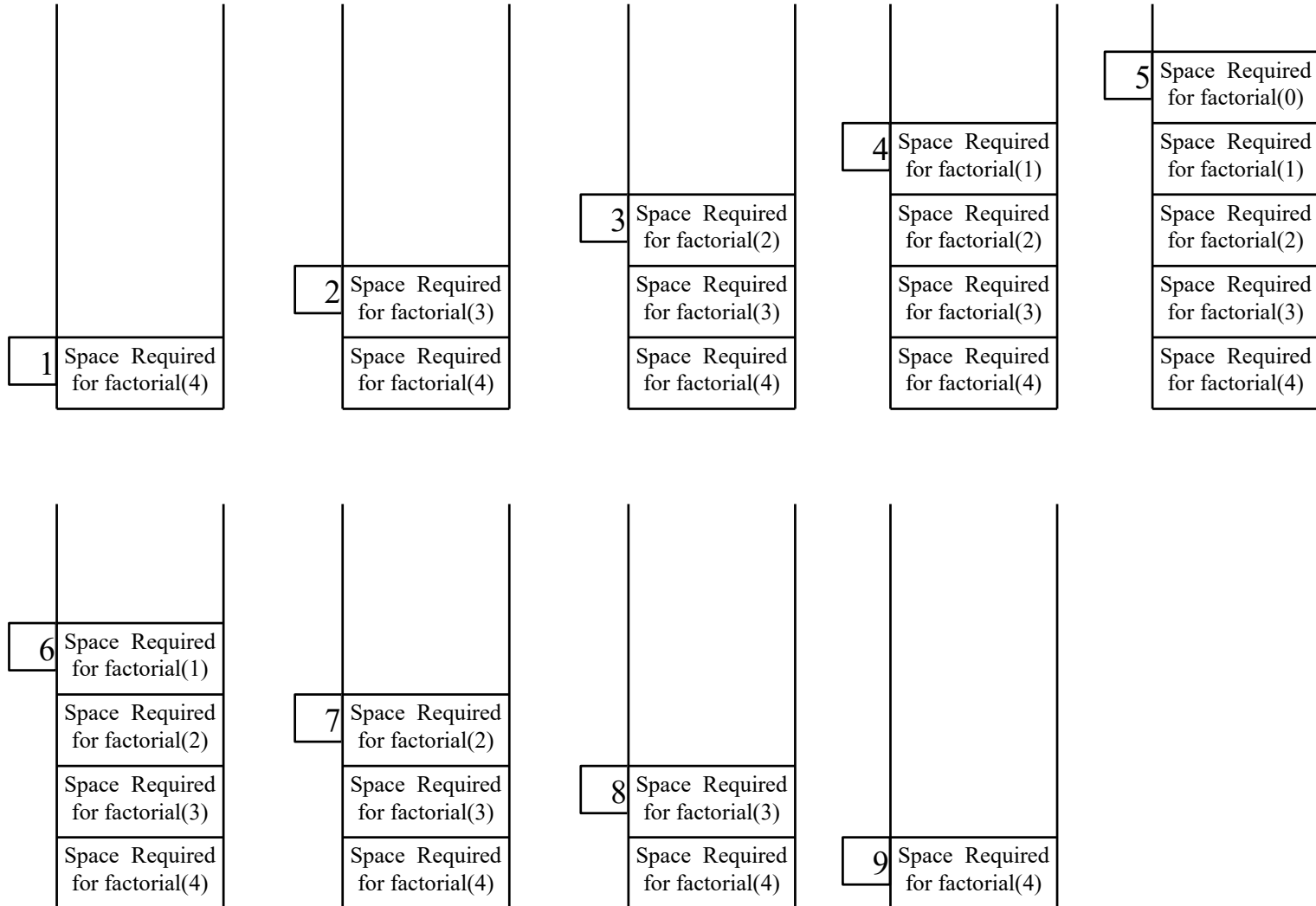
Trace code



Trace code



Trace stack



Stack overflow

- Deep recursion may result in stack overflow
- If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, *infinite recursion* can occur

- Example

```
public static long factorial(int n) {  
    // Mistakenly omit base case  
    return n * factorial(n - 1);  
}
```

- Results in stack overflow

Computing factorials

- As a recursive method

```
public static long factorial(int n) {
    if (0 == n) {
        // Base case
        return 1;
    }
    else {
        // Recursive call
        return n * factorial(n - 1);
    }
}
```

Direct recursion



A recursive method is one that invokes itself directly or indirectly

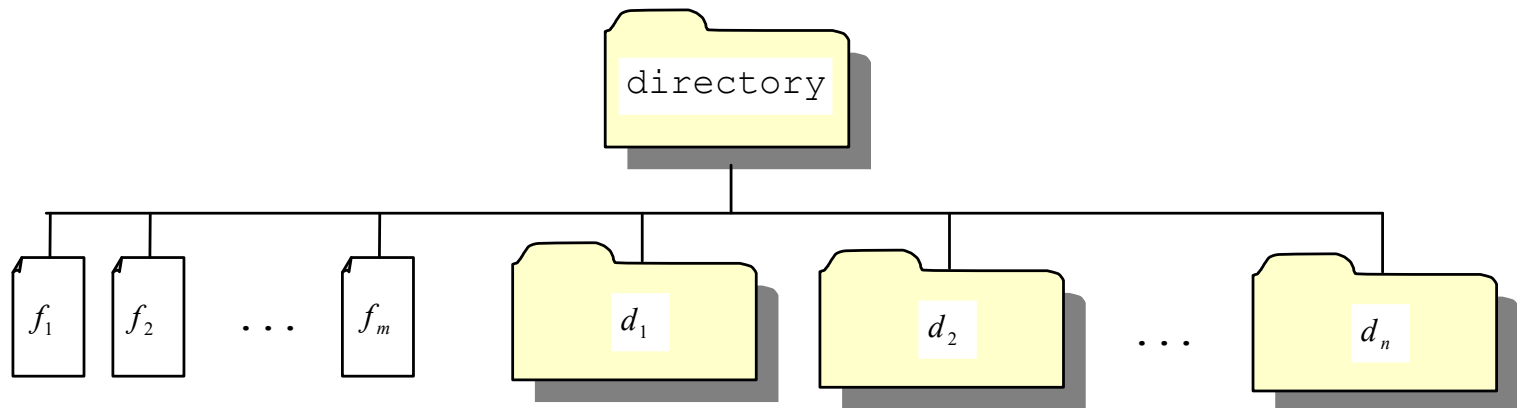
- As a non-recursive method

```
public static long factorial(int n) {
    long nfactorial = 0 == n ? 1 : n;
    for (int i = n - 1; 1 < i; --i) {
        nfactorial *= i;
    }
    return nfactorial;
}
```

Recursive algorithms can be replaced with non-recursive counterparts. However, some problems are inherently recursive, and difficult to solve without using recursion.

Recursion in practice

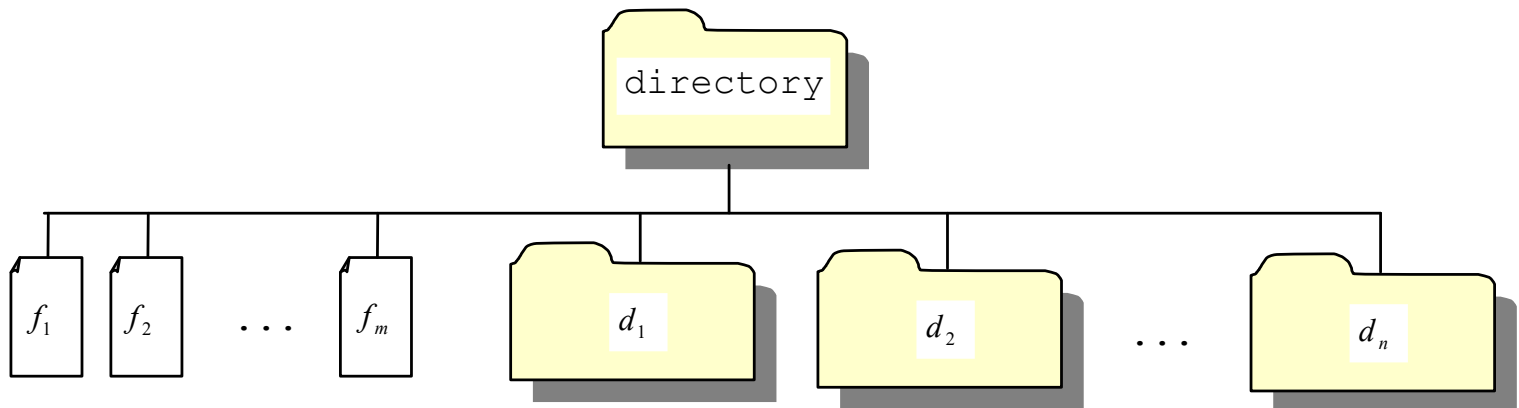
- In practice, recursive methods are used to efficiently solve problems with recursive structures
 - Example problem: find the size of a directory



Finding the directory size

- The size of a directory is the sum of the sizes of all files in the directory
- A directory may contain subdirectories
- Suppose a directory contains files and subdirectories
- The size of the directory can be defined recursively as

$$\text{size}(d) = \text{size}(f_1) + \text{size}(f_2) + \dots + \text{size}(f_m) + \text{size}(d_1) + \text{size}(d_2) + \dots + \text{size}(d_n)$$



Characteristics of recursion

- All recursive methods have the following characteristics
 - The method is implemented using an if-else (or a switch) statement that leads to **different cases**
 - One or more **base cases** (the simplest case) are used to stop recursion
 - Every recursive call **reduces** the original problem, bringing it increasingly **closer to a base case** until it becomes that case
- In general, to solve a problem using recursion, you break it into subproblems
 - If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively
 - This subproblem is almost the same as the original problem in nature with a smaller size

Recursion vs. iteration

- Recursion is an alternative form of program control
- It is essentially repetition without a loop
- Recursion bears substantial overhead
 - Each time the program calls a method, the system must assign space for all of the method's local variables and parameters
 - This can consume considerable memory and requires extra time to manage the additional space

Recursion vs. iteration

- Recursive algorithms can be replaced with non-recursive counterparts
 - If performance is a concern, then avoid using recursion
 - However, some problems are inherently recursive, and difficult to solve without using recursion
- Use whichever approach can best develop an intuitive solution that naturally mirrors the problem
 - If an iterative solution is obvious, then use it

Next Lecture

- Arrays