SE 2205a: Data Structures and Algorithm for Object Oriented Design (OOD)

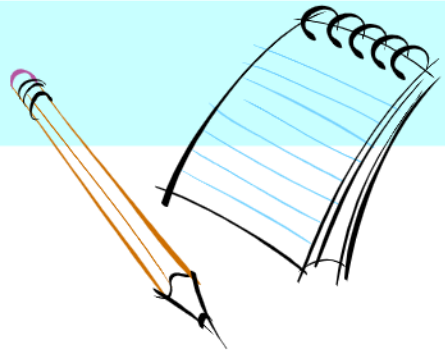
Unit 1 – Part 1: Tying the loose ends in Java – Recursion and More

Dr. Quazi M. Rahman, Ph.D, P.Eng, SMIEEE
Office Location: TEB 263
Email: QRAHMAN3@uwo.ca
Phone: 519-661-2111 x81399

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

"Genius is 1% talent and 99% percent hard work..." ~ Albert Einstein

Outline



- ❑Reviewing Java Fundamentals
- ❑Introducing Wrapper Class
- ❑Introducing Bitwise operators
- ❑Introducing Recursion

Review

1) A computer program is the implementation of an algorithm

   a) True
   b) False

Review

2) An algorithm refers to

a) The collection of instructions that the computer can understand
b) A code that allows us to type in text materials
c) A set of math equations to derive the solution of a problem
d) A step-by-step solution to solve a specific problem

Review

3) The extension of a java source code file is

a) .java
b) .obj
c) .class
d) .exe
e) .cpp

Review

4) True or False: The source code in Java is compiled into a bytecode.

   a) True
   b) False

Review

5) True or False: Bytecode is machine dependent.

a) True
b) False

Review

6) In the following java statement what does `System` represent?

```
System.out.println("Welcome to Java!");
```

a) A package
b) A class
c) An object
d) A method
e) None of the above

Review

7) True or False: In Java programming language, the main() method must be declared as a static method of a class.

   a) True
   b) False

Review

8) Which of the following statements is FALSE?

a) In Java every class belongs to a class.

b) In Java, the entry point for the java interpreter is the main() method.

c) In Java, a class definition should always be terminated with a semicolon.

d) println() is a PrintStream type method.

Review

9) What will be the value of 'c' after the third statement? (Let's draw the memory diagram to find our answer; in class discussion)

```
int a = 3, b = 2;
double c;
c = (a+b)/2;
```

a) 2.5
b) 2.0
c) 3.0
d) 0

Review

10) What will be the value of 'c' after the third statement? (Let's draw the memory diagram to find our answer; in class discussion)

```
float a = 3, b = 2;
int c;
c = (a+b)/2;
```

a) 2.5
b) 2
c) 0
d) The code will not compile

Review

11) What will be the output of the code? Let's draw the memory diagram (in class discussion)

```
int a = 2, b = 4;
 a = ++b;
 System.out.println(a+", "+b);
```

a) 2, 5
b) 4, 4
c) 5, 5
d) 4, 5

Review

12) Output ?

```
int a = 2; float b = 4.0;
System.out.println(a/b);
```

a) 0
b) 0.5
c) The code will result in a Compilation error
d) The code will result in a Run time error

Primitive Data Types VS. Wrapper (Class) Reference Types

- There are many data structures and algorithms in Java's libraries that are specifically designed so that they only work with reference type data instead of primitive types.
- To get around this obstacle, Java defines a **wrapper** class for each base type.
  - Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic **boxing** (primitive type to object-reference type) and **unboxing** (object-reference type to primitive type) .

Example: Wrapper (Class) reference Types

Base Type	Wrapper Class name	Example	Access Example
boolean	Boolean	Boolean b = true;	System.out.println(b.booleanValue());
char	Character	Character c = 'a';	System.out.println(c.chracterValue());
byte	Byte	Byte bt = (byte)34;	System.out.println(bt.byteValue());
short	Short	Short s = (short)100;	System.out.println(s.shortValue());
int	Integer	Integer a = 5;	System.out.println(a.intValue());
long	Long	Long lng = 1089L;	System.out.println(lng.longValue());
float	Float	Float f = 2.37F;	System.out.println(f.floatValue());
double	Double	Double d = 4.3;	System.out.println(d.doubleValue());

Note: all the above can be printed directly without the …value() method

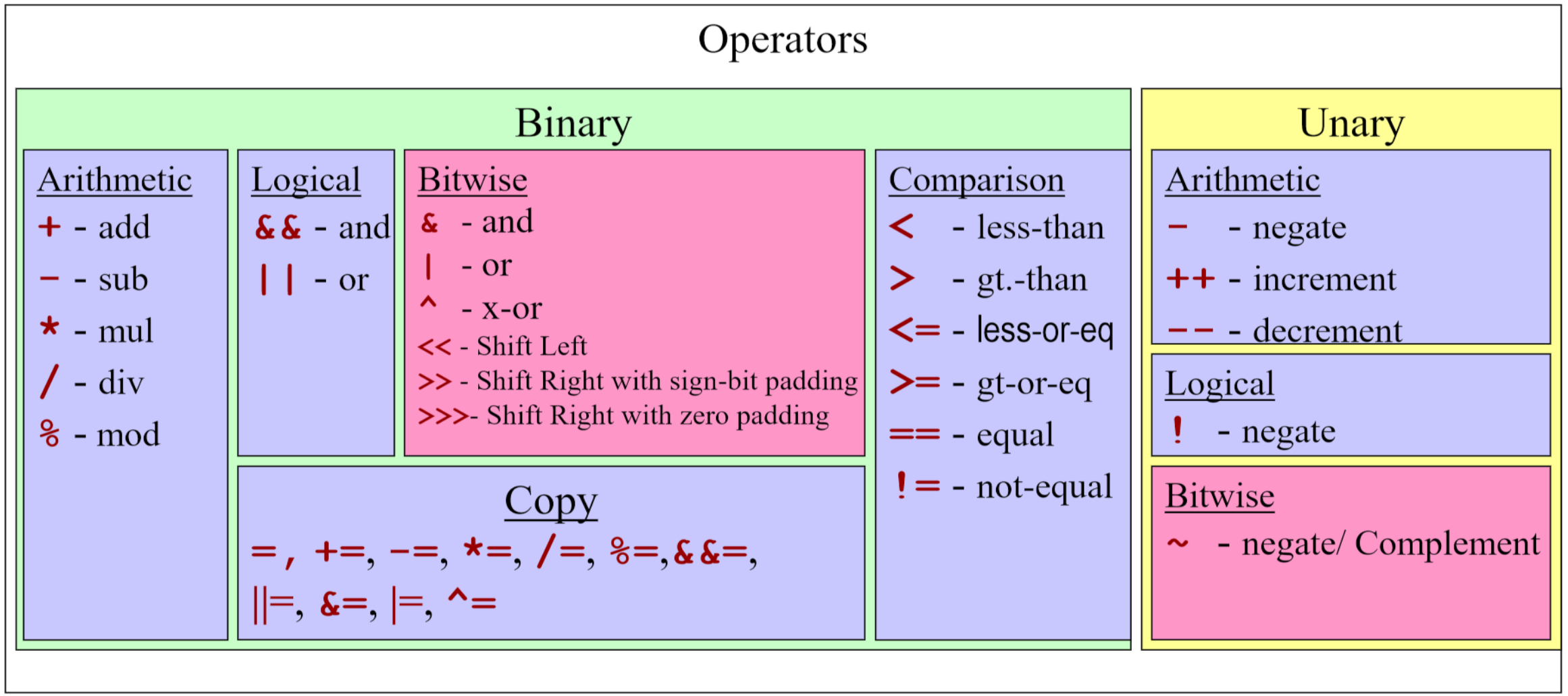
Code-Example:

```
Integer a = 5; // boxing / autoboxing – integer 5 is boxed/ wrapped inside Integer 'a'
int b = a; //unboxing – Integer 'a' is unboxed and assigned to int 'b'.
int k = Integer.parseInt("34"); //using static method parseInt() from Integer class
```

Note on Wrapper (Class) reference Types

- Any two numerical wrapper classes cannot be assigned to each other, although it is possible for primitive type data.
  - Integer a = 5;
  - Double x = a; //Not valid
  - Double y = 4; //Not valid; it must be Double y = 4.0
  - double m = 2; Double n = m; //valid

Java Operator Map



Bitwise Operators

- Java provides the following bitwise operators for integers and Booleans (see the example on the next slide):

~	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit
>>>	shift bits right, filling in with zeros

Examples with Bitwise Operators and use of Wrapper-Class methods

```
public class DemoBitwiseOperators {
   public static void main(String[] args) {
      int x = 9, y = 10, s = 3;
      System.out.printf("%d in Binary is: %s\n",x,Integer.toBinaryString(x));
      System.out.printf("%d in Binary is: %s\n",y,Integer.toBinaryString(y));
      System.out.printf("Binary OR: %d | %d = %d, and in Binary: %s\n",x,y, x|y,Integer.toBinaryString(x|y));
      System.out.printf("Binary AND: %d & %d = %d, and in Binary: %s\n",x,y, x&y,Integer.toBinaryString(x&y));
      System.out.printf(" Binary EX-OR: %d ^ %d = %d, and in Binary: %s\n",x,y, x^y,Integer.toBinaryString(x^y));
      System.out.printf("Complement of %d in Binary (32 bits) is: %s\n",x,Integer.toBinaryString(~x));
      System.out.printf("Shifting %d left by %d bits; in Binary is: %s\n",x,s,Integer.toBinaryString(x<<s));
   }
}
```

Note: One can also use toOctalString() or toHexString() methods to convert an integer to either Octal number or Hexadecimal number.

Output in 32 bits

```
9 in Binary is: 1001
10 in Binary is: 1010
Binary OR: 9 | 10 = 11, and in Binary: 1011
Binary AND: 9 & 10 = 8, and in Binary: 1000
Binary EX-OR: 9 ^ 10 = 3, and in Binary: 11
Complement of 9 in Binary (32 bits) is: 11111111111111111111111110110
Shifting 9 left by 3 bits; in Binary it is: 1001000
```

Introduction to Recursion

- We are used to calling other methods from a method in Java (or any other programming language).
- It's also possible for a method to call itself.
- A method that calls itself is a *recursive method*.
- Example: EndlessRecursion.java

```
public class EndlessRecursion {
    public static void main(String[] args) {
        message();
    }
    public static void message() {
        System.out.println("Do Nothing!");
        message();
    }
}
```

Introduction to Recursion

- This method in the example displays the string “This is a recursive method.”, and then calls itself.
- Each time it calls itself, the cycle is repeated.
- Like a loop, a recursive method must have some way to control the number of times it repeats. Here, the variable n is the controlling variable.

```
public class RecursionDemo {
    public static void main(String[] args) {
        int n = 3;
        message(n);
    }
    public static void message(int n)
    {
        if(n>0) {
            System.out.println("This is a recursive method.");
            message(n - 1);
        }
    }
}
```

Endless Recursion: No controlling variable!

```
public class EndlessRecursion {
    public static void main(String[] args) {
        message();
    }
    public static void message() {
        message();
    }
}
```

Solving Problems With Recursion: The facts

- Recursion can be a powerful tool for solving repetitive problems.
- Recursion is NEVER absolutely required to solve a problem.
- Any problem that can be solved recursively can also be solved iteratively, with a loop.
- In many cases, recursive algorithms are less efficient than iterative algorithms (discussed later).
- Generally Recursive algorithms use Top-Down approach while the iterative algorithms use Bottom-Up approach.
- In top-down approach, we try to figure out how to divide the problem into sub-problems, and then find the solution. In the bottom-up approach, we start to solve the problem for a simple case. Then we figure out how to solve the bigger versions.

Solving Problems With Recursion: The Overhead

- Recursive solutions (top-down) uses memory repetitively by….
  - allocating memory for parameters and local variables, and
  - storing the address of where control returns after the method terminates.
- These actions are called *overhead* and take place with each method call.
- This overhead does not occur with iterative approach (bottom-up) that uses a loop to table the result after each iteration.
- Some repetitive problems are more easily solved with recursion than with iteration.
  - Iterative algorithms might execute faster; however,
  - a recursive algorithm might be designed faster.

Content of a Recursive Method

- Base case(s)
  - Values of the input variables, for which we perform no recursive calls, are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
  - There can be more than one base-case for a recursive method.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.
  - In a recursive method-call, the number of times a method calls itself is known as the depth of recursion.

Iteration vs. Recursion

❑ Iteration

Iteration is generally a bottom-up technique used to run a block of code repeatedly until a specific condition no longer exists. Iterations are normally performed with loops.

❑ Recursion

Recursion is generally a top-down technique that repeats itself until a base case, in terms of a simpler version of itself, is satisfied.

Recursive vs. Iterative: An Example

The factorial of a positive integer "n" is the product of all positive integers less than or equal to n :

$$n! = n * (n-1) * (n-2) * \ ... * 2 * 1$$

The above expression can be solved iteratively as follows:

```
public static double factorialIterative (int n) {
        double result=1; int i=1; //bottom-up: result is calculated for a simple case
            while (i<=n) {
                result = result*i;
                i++;
             }
            return result;
        }
```

Recursive vs. Iterative: An Example

The factorial of a positive integer "n" can be calculated recursively, as follows:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

```
public static double factorialRecursive(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorialRecursive(n-1); /*Recursive call =>top-down approach divides
                  the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3)

$$factorial(0) = 1;$$
$$factorial(n) = n*factorial(n-1);$$

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for
which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down
approach divides the problem into sub-problems and
involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3) = 3 * factorial(2)

$$factorial(0) = 1;$$
$$factorial(n) = n*factorial(n-1);$$

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for
which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down
approach divides the problem into sub-problems and
involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down approach divides the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down approach divides the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * (2 * (1 * 1)))

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down approach divides the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * (2 * (1 * 1)))

= 3 * (2 * 1)

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down approach divides the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

Computing Factorial

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * (2 * (1 * 1)))

= 3 * (2 * 1)

= 3 * 2

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down approach divides the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

Computing Factorial

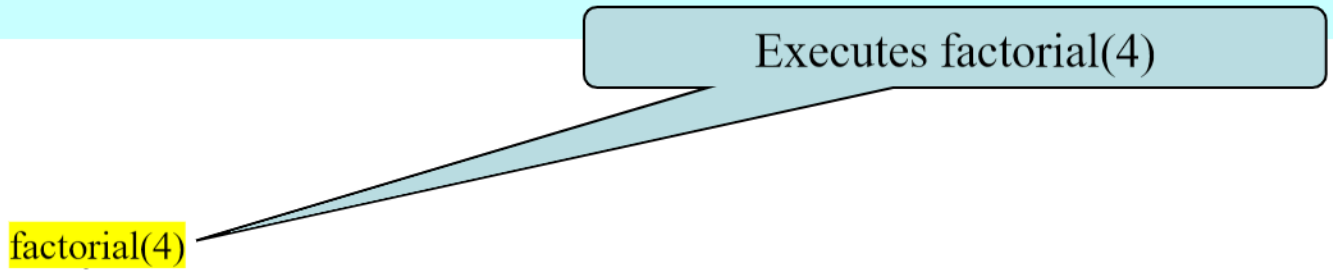
factorial(3) = 3 * factorial(2)
= 3 * (2 * factorial(1))
= 3 * (2 * (1 * factorial(0)))
= 3 * (2 * (1 * 1)))
= 3 * (2 * 1)
= 3 * 2
= 6

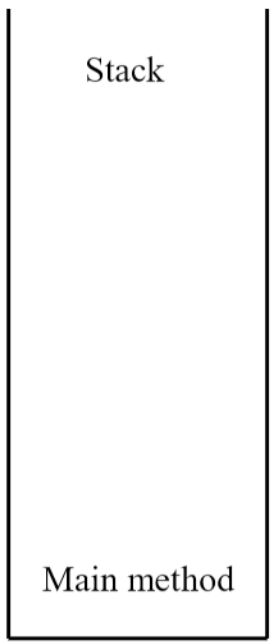
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

```
public static double factorial(int n){
  if(n == 0)  { //base case: value of the input variable, for which we stop making any call
      return 1;
  }
  else  {
      return n*factorial(n-1); /*Recursive call =>top-down approach divides the problem into sub-problems and involves memory for each case*/
   }
}
```

animation

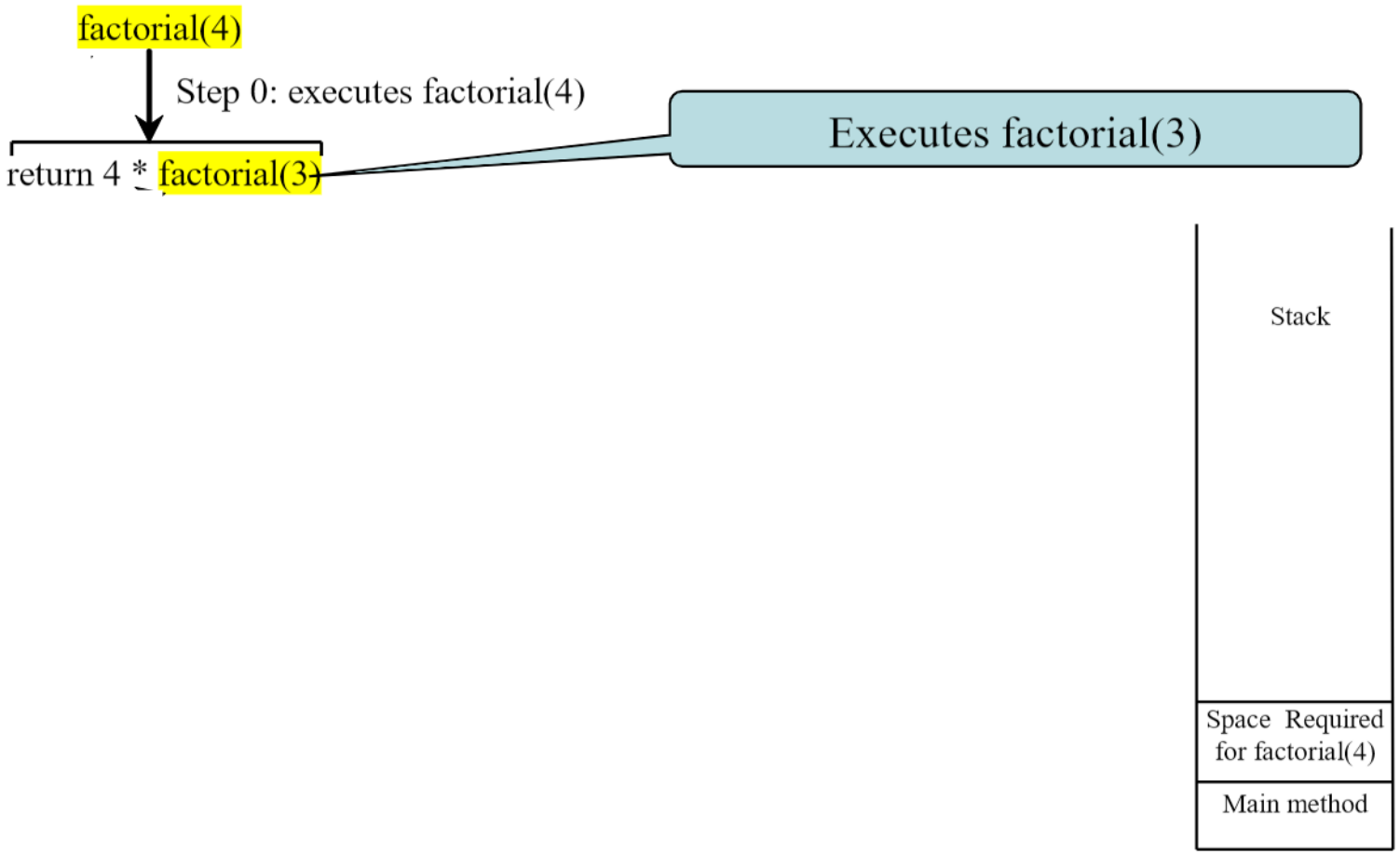
Trace Recursive factorial





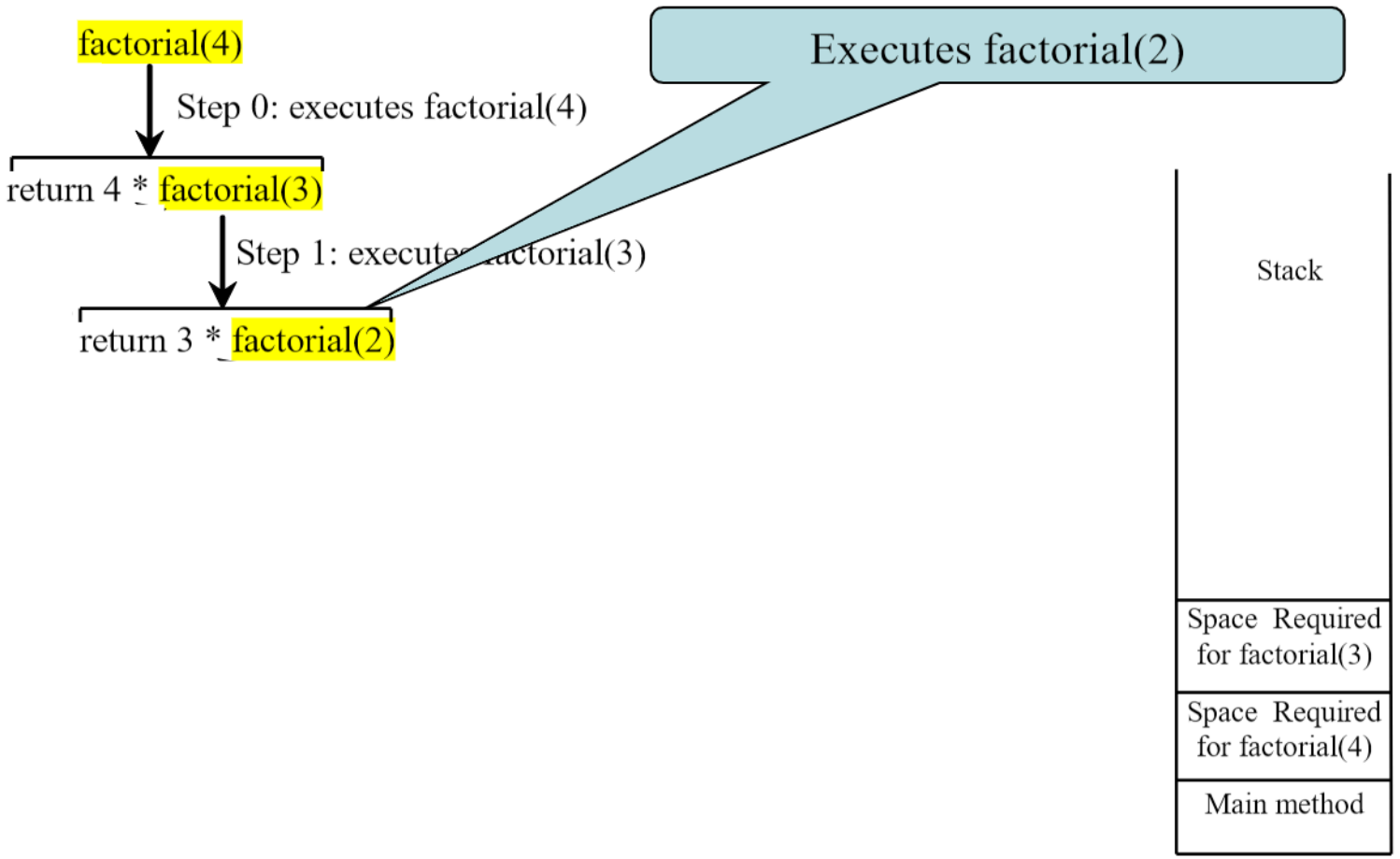
animation

Trace Recursive factorial



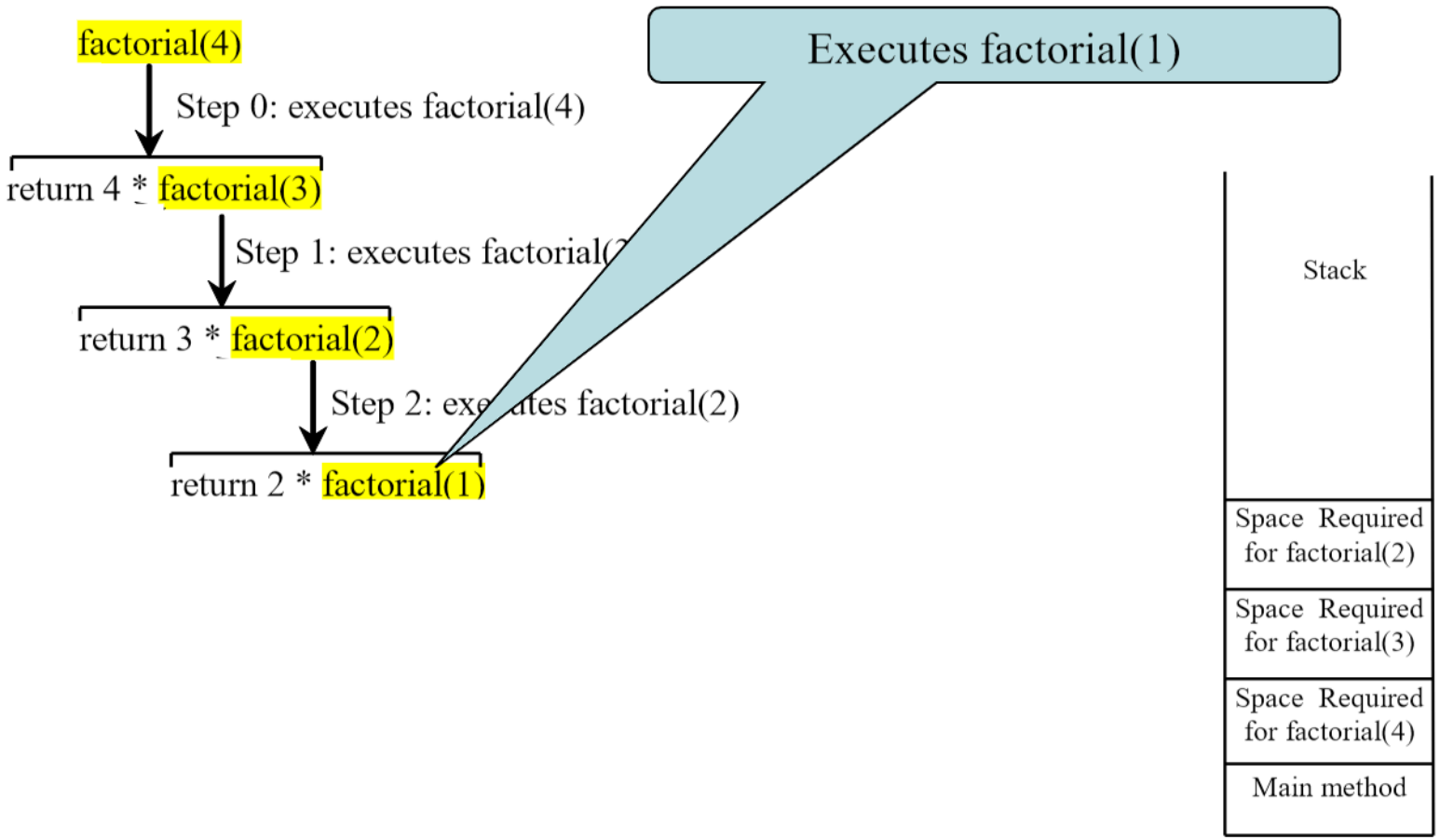
animation

Trace Recursive factorial



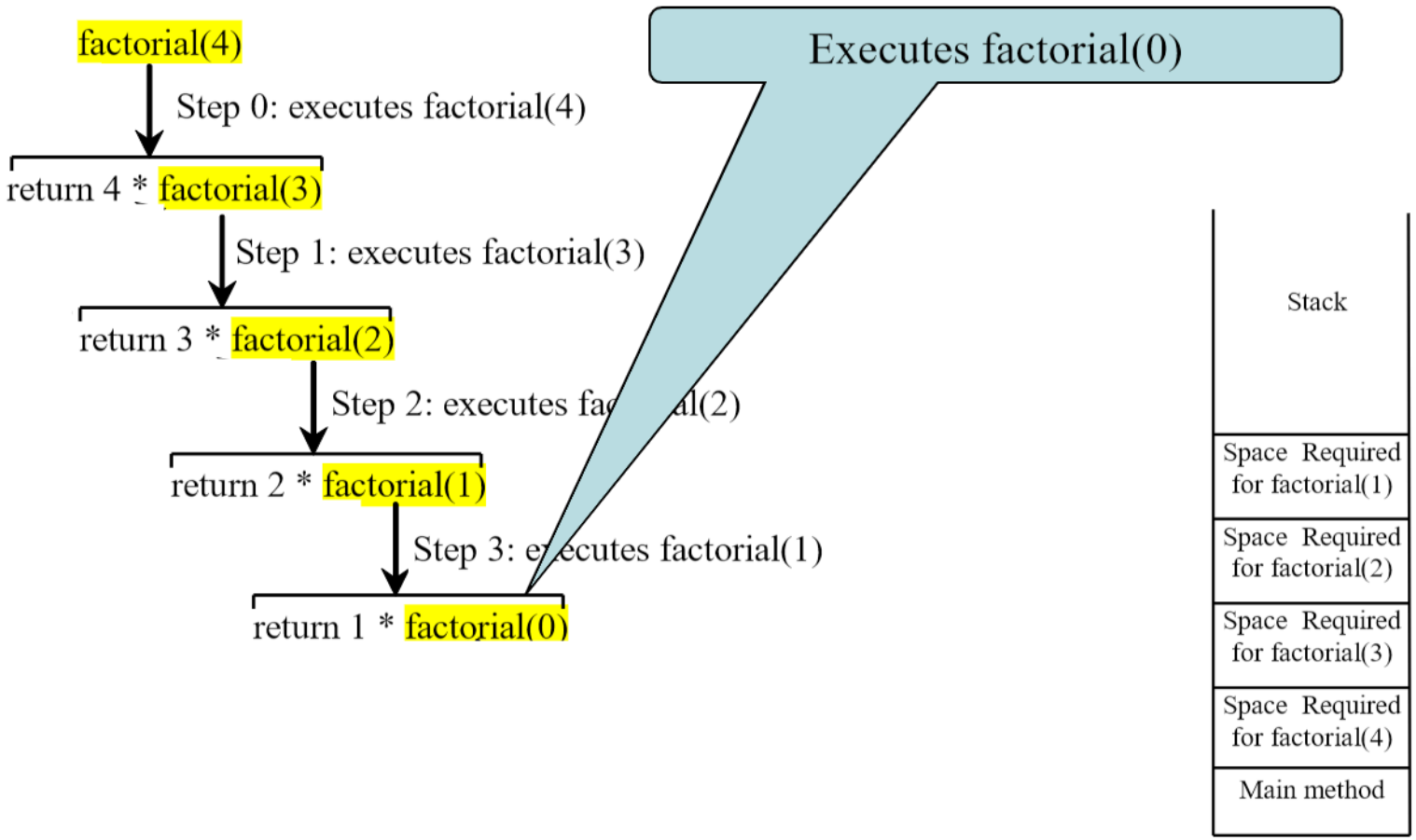
animation

Trace Recursive factorial



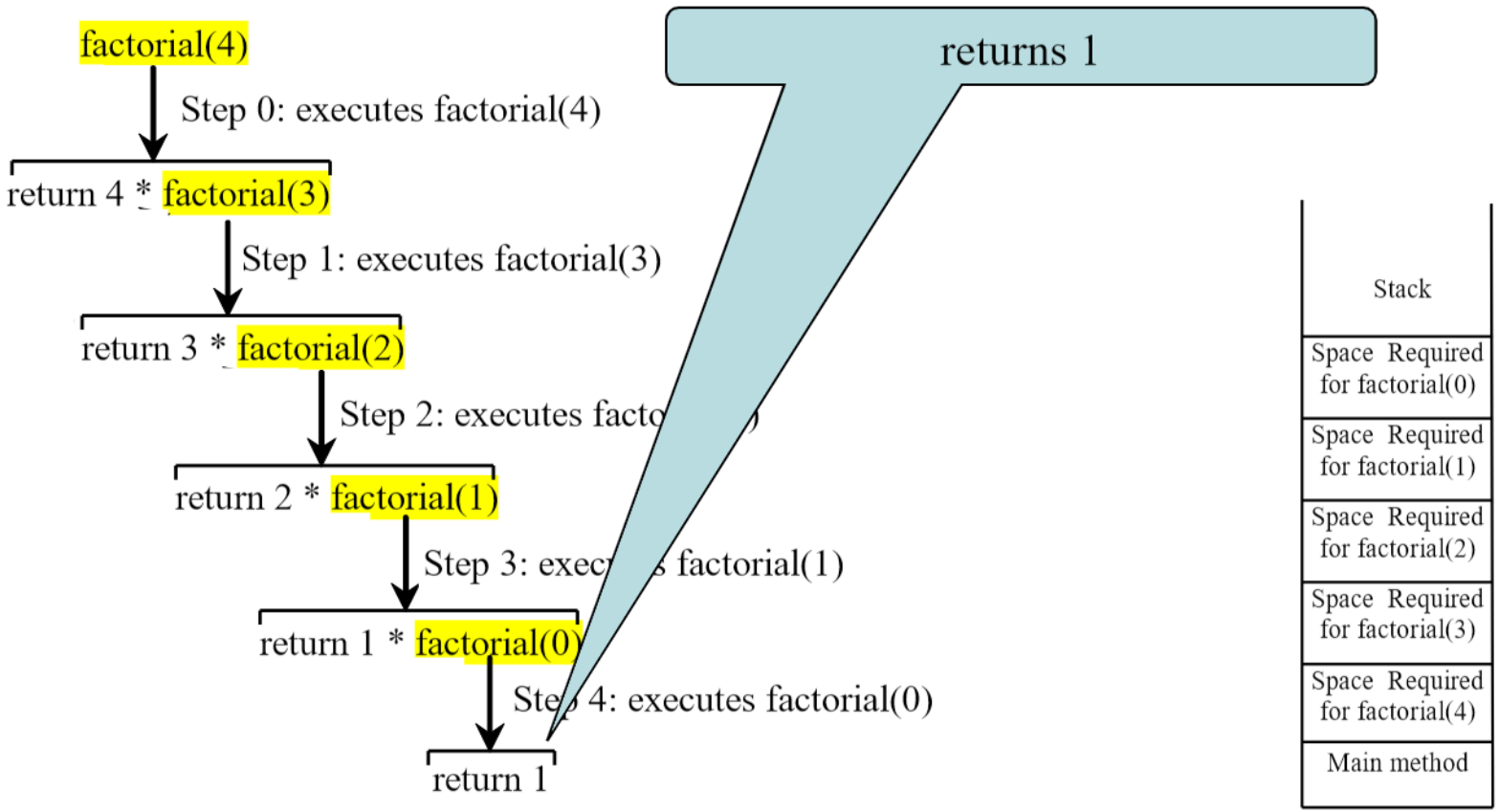
animation

Trace Recursive factorial



animation

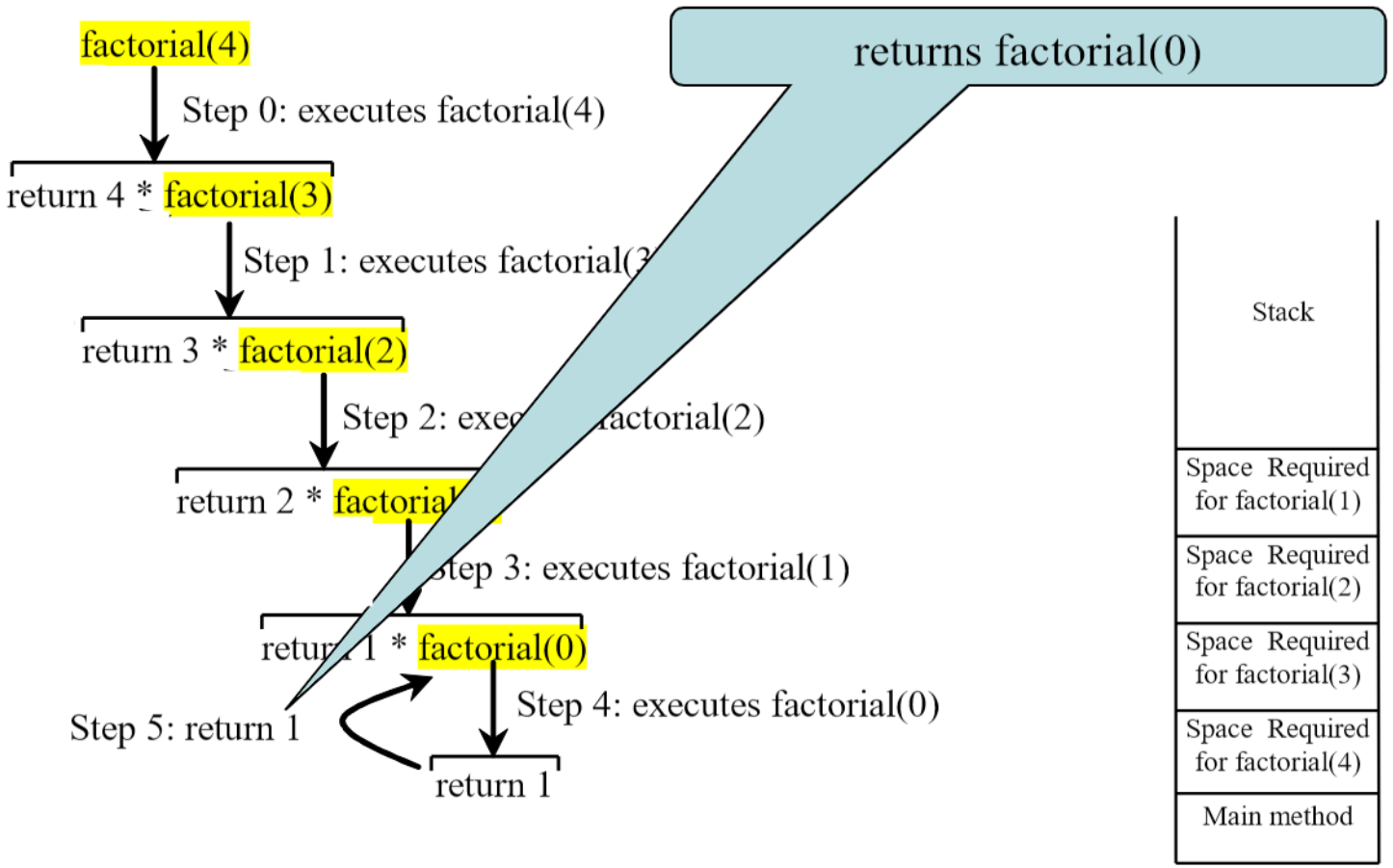
Trace Recursive factorial



Depth of recursion is 4, that is the method is calling itself for 4 times

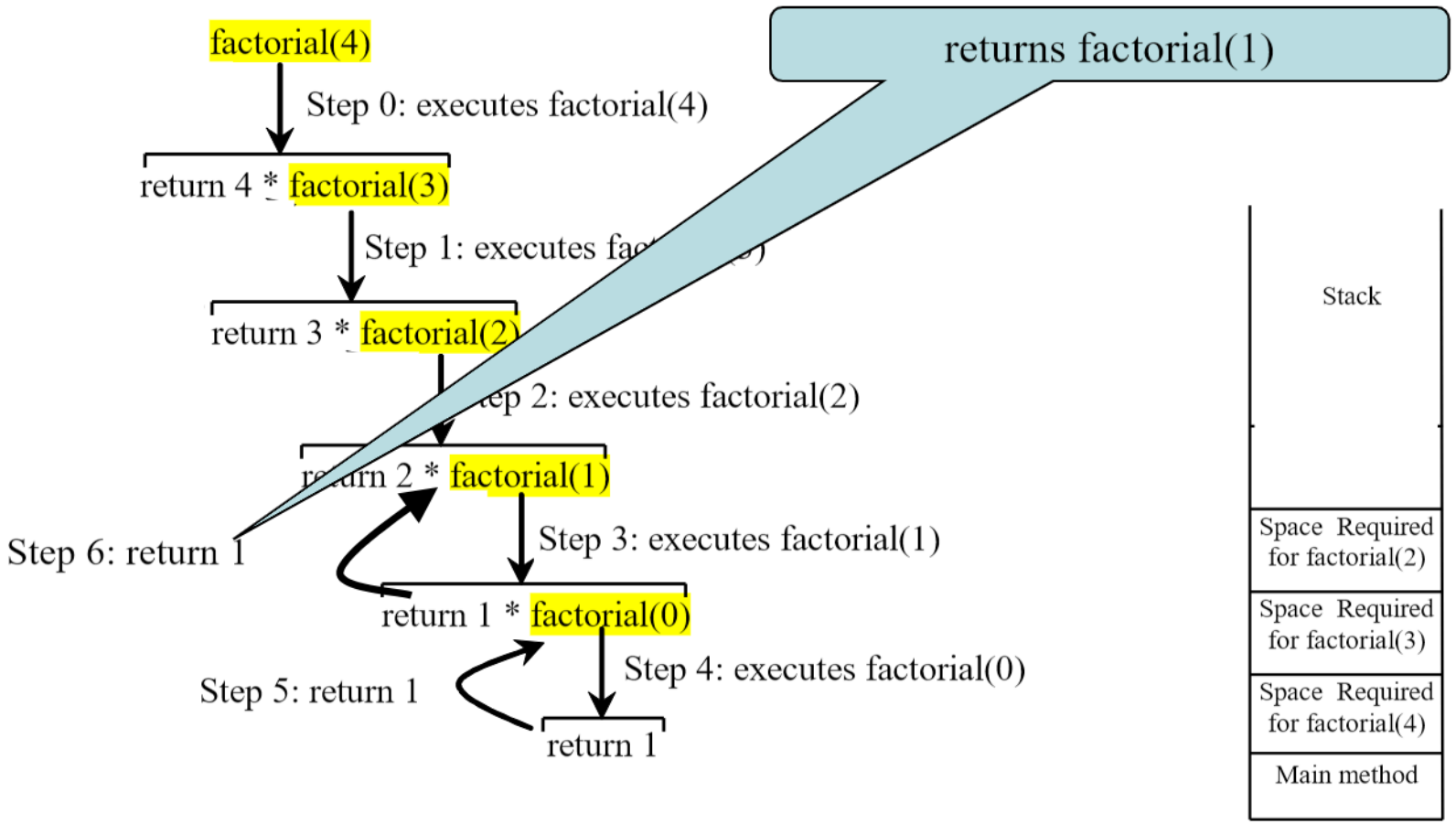
animation

Trace Recursive factorial



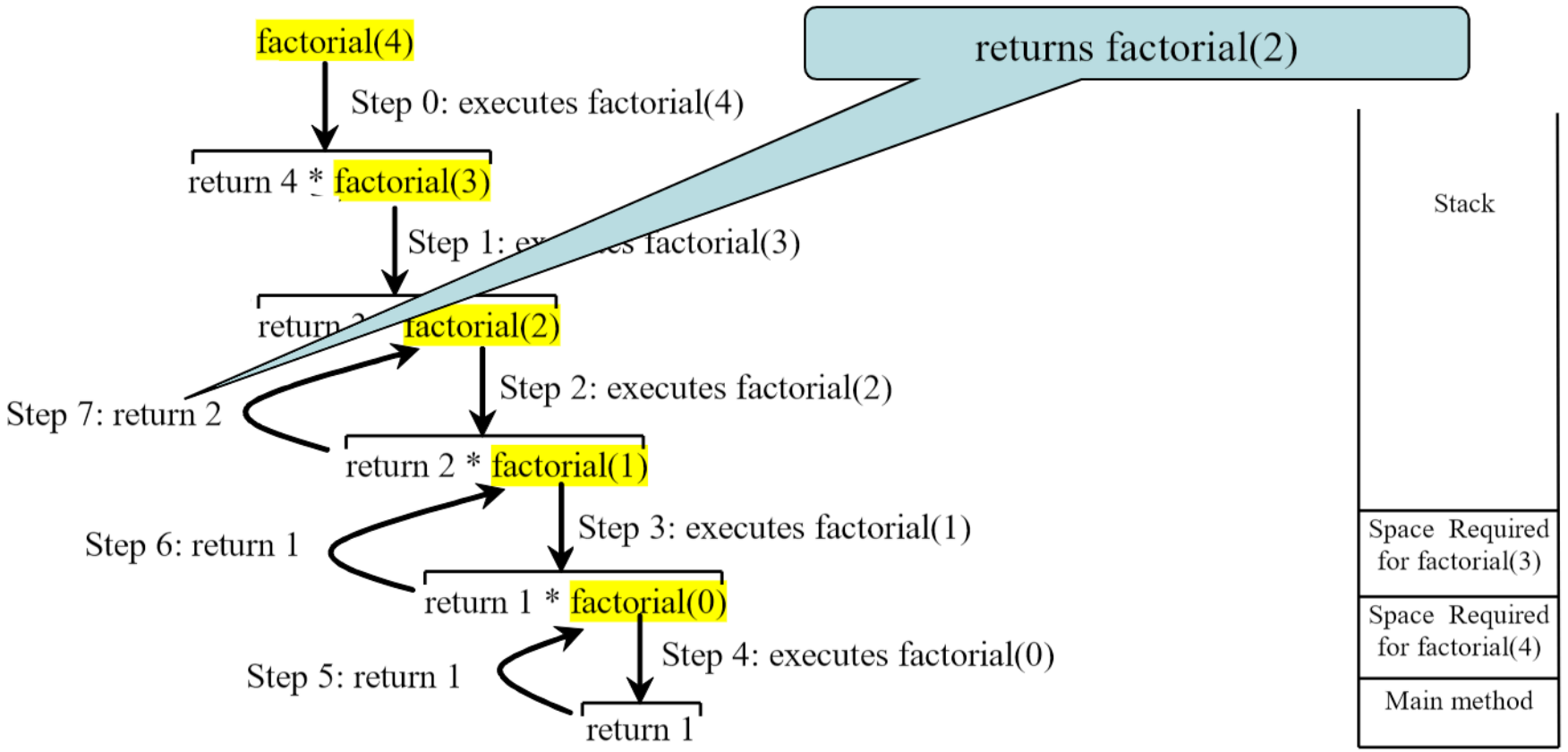
animation

Trace Recursive factorial



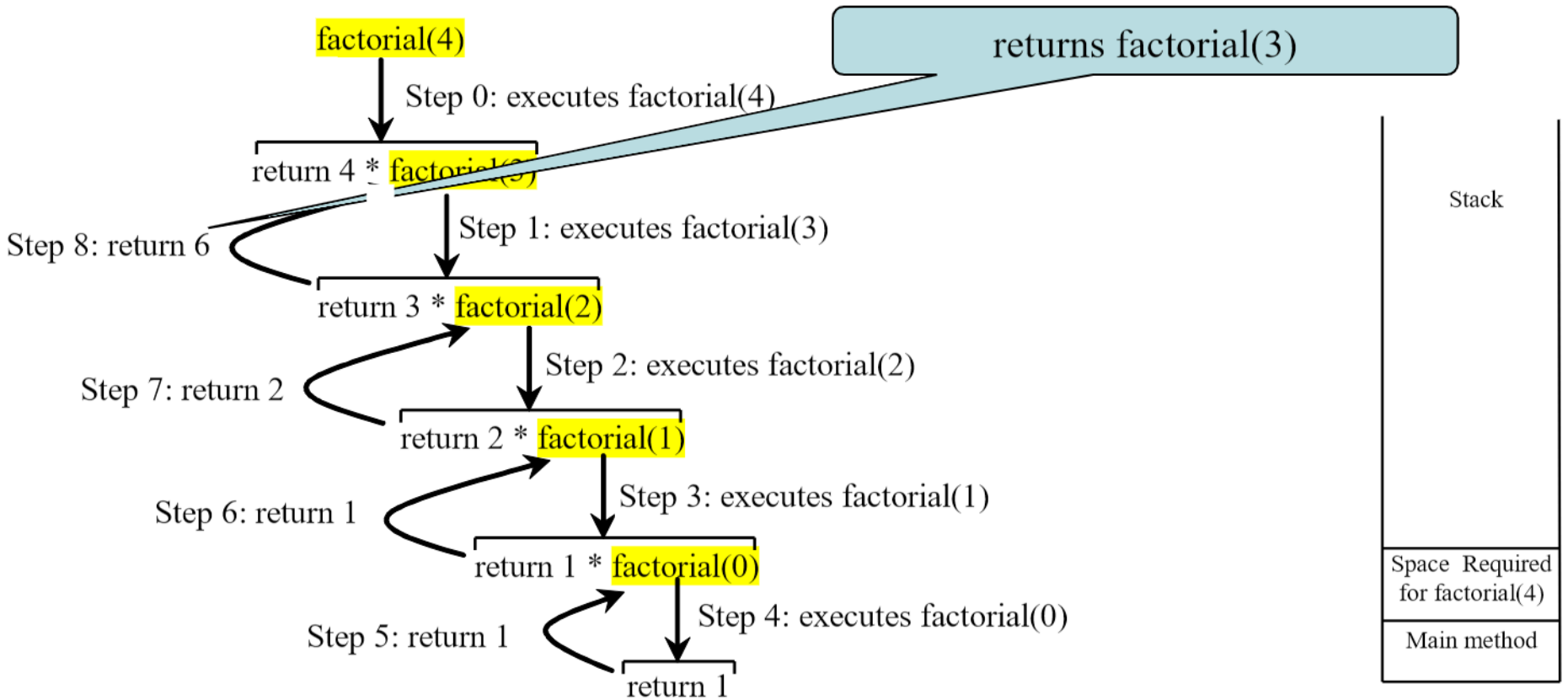
animation

Trace Recursive factorial



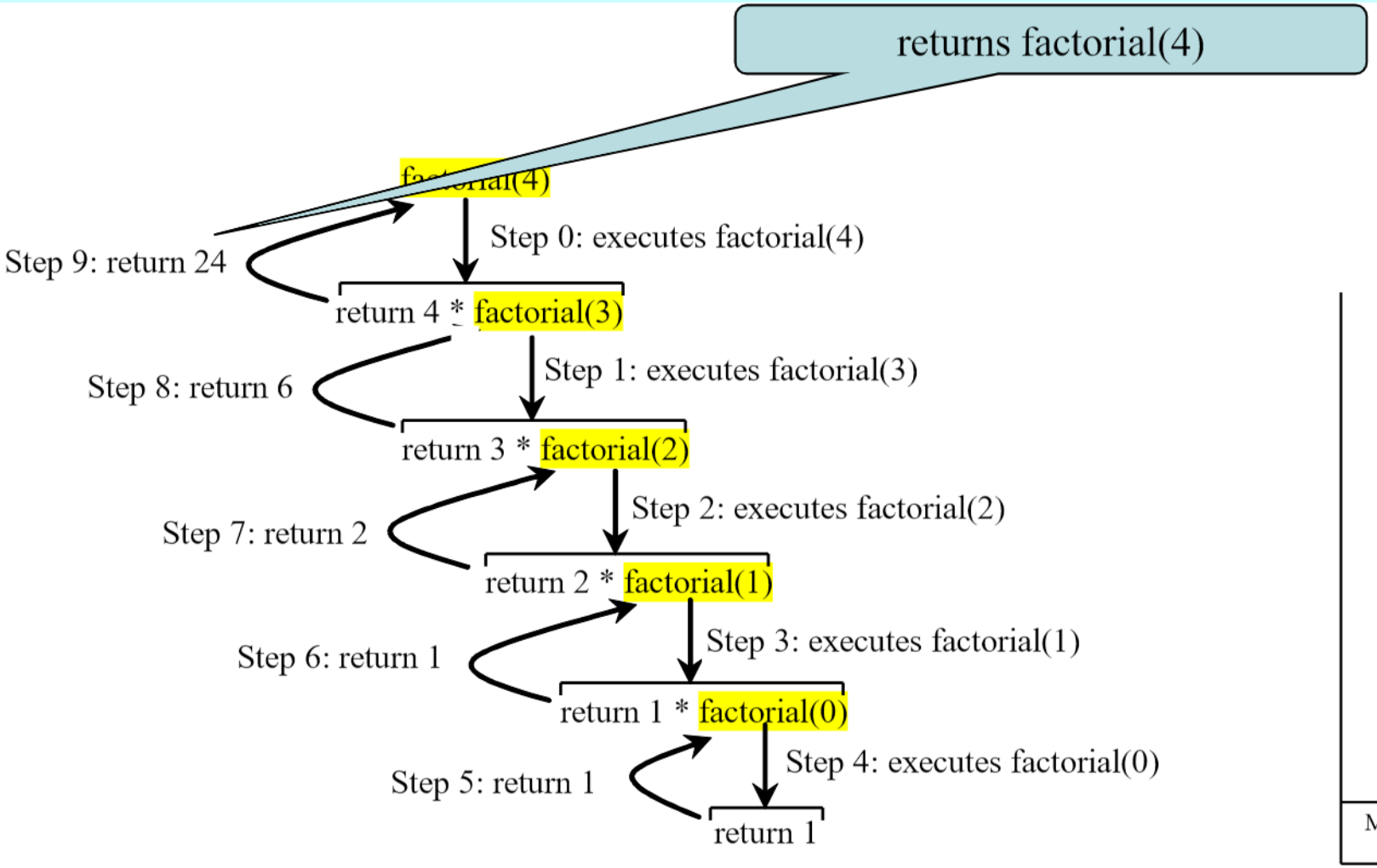
animation

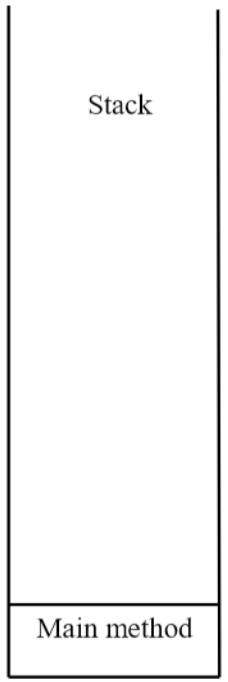
Trace Recursive factorial



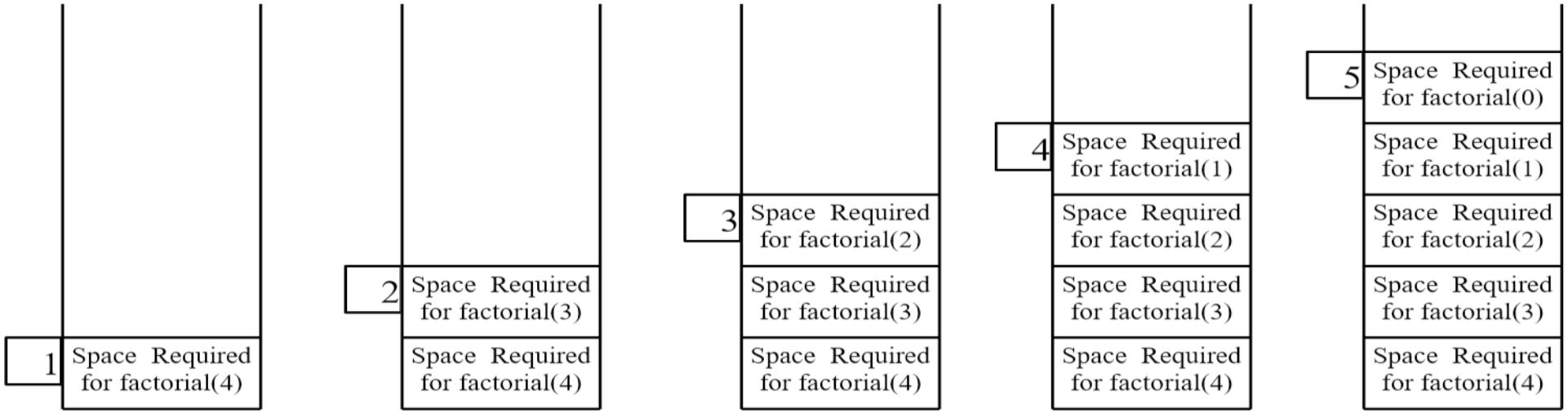
animation

Trace Recursive factorial

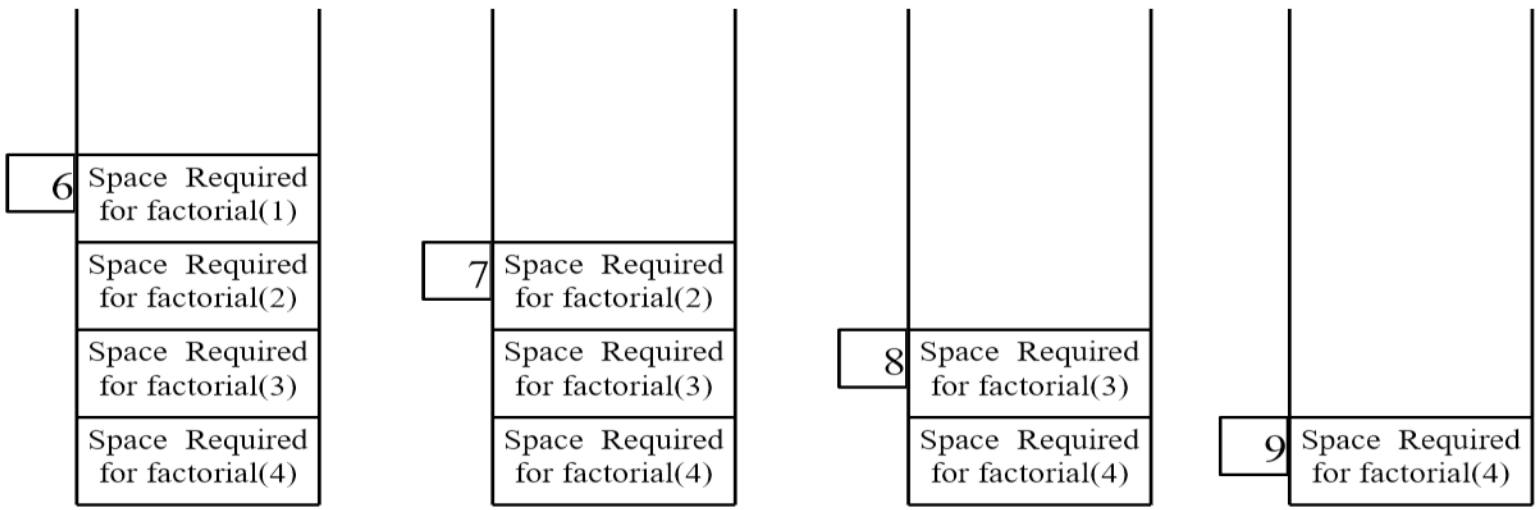




factorial(4) Stack Trace



Depth of recursion is 4, that is the method is calling itself for 4 times



Review

13) What is the advantage of recursive approach than an iterative approach?

a) Consumes less memory
b) Easy to implement if the base case is known, and Less code, in general
c) Consumes more memory
d) More code must be written

Exercise 1

- Write a program to find out the factorial of 30 by using both iterative approach and recursive approach and see the code-execution time-difference between these two by using the System.nanoTime() method in your code appropriately.
  - nanoTime() method header: public static long nanoTime()

  It returns the current value of the most precise available system timer, in nanoseconds. This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds (1 Nano second = 1 x 10^{-9} seconds) since some fixed but arbitrary *origin* time (perhaps in the future, so values may be negative).
- To find out the date and time we need to use System.currentTimeMillis() (Read the Javadoc description) which can also be used to find the elapsed time, but the accuracy is little less when compared to nanoTime().
- Note: If you write the iterative loop in the main() method, the iterative time will be less than recursive time but, if you write the iterative loop inside another method you need to check the time inside that method instead of main(), otherwise, the interactive approach will take more time. The simple explanation is – the method that calls itself results in a big overhead in processing time. Sometimes, it has been found that in functional programming language implementations (programs that are implemented using methods), iteration can be very expensive, and recursion can be very cheap.

FYI: Code for finding current date and time using System.currentTimeMillis()

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class CurrentDateAndTime {
    public static void main(String[] args) {
        long yourmilliseconds = System.currentTimeMillis();
        SimpleDateFormat sdf = new SimpleDateFormat("MMM dd,yyyy HH:mm");
        Date resultdate = new Date(yourmilliseconds);
        System.out.println(sdf.format(resultdate));
    }
}
```