AP Computer Science B

Java Object-Oriented Programming [Ver. 3.0]

Unit 5: Algorithms

CHAPTER 17A: RECURSION

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- What is recursion? Repeated Pattern
- Base-Condition and Recursive Formula
- Recursive Programs
- Fractals (Application of Recursion)
- Tail Recursion
- Fibonacci Numbers (Iterative/Recursive Versions)

Overview

LECTURE 1

Sequence

- A sequence [1, 2, 3, 4, 5, …, n] can be defined as

  $a_1 = 1$,

  $a_2 = 2$,

  $a_3 = 3$,

  … ,

  $a_n = n$, (Explicit formula)

Sequence

- A sequence [1, 2, 3, 4, 5, …, n] can be defined as

  $a_1 = 1,$

  $a_2 = a_1+1,$

  $a_3 = a_2+1,$

  $\dots ,$

  $a_n = a_{n-1}+1,$ (Recursive formula)

  When data or functions are defined by their own sub-cases, we call it recursive definition.

Recursive Definition

- Recursive Formula is widely used in sequence or series definition in Mathematics.

Syntax diagram

- Syntax diagrams (or railroad diagrams) are a way to represent a context-free grammar.
- They represent a graphical alternative to Backus–Naur form, EBNF, Augmented Backus–Naur form, and other text-based grammars as metalanguages.

Context-Free Grammar

Context-free grammars have the following components:

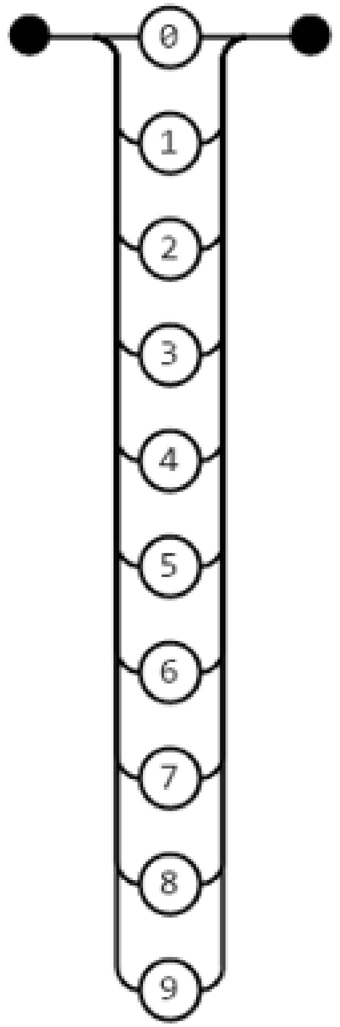
- **A set of terminal symbols** which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
- **A set of nonterminal symbols** (or variables) which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
- **A set of production rules** which are the rules for replacing nonterminal symbols. Production rules have the following form: variable → string of variables and terminals.
- **A start symbol** which is a special nonterminal symbol that appears in the initial string generated by the grammar.

Context-Free Grammar

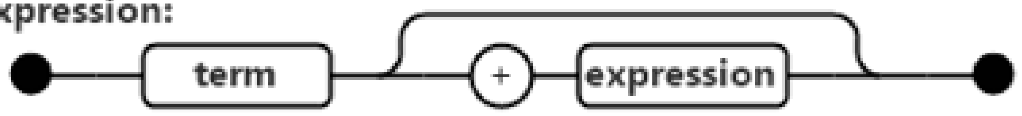
Here is a context-free grammar that generates arithmetic expressions (subtraction, addition, division, and multiplication)[1].

- Start symbol = <expression>
- Terminal symbols = {+,−,∗,/,(,),number}, where “number” is any number
- Production rules:
  - <expression> → number
  - <expression> → (<expression>)
  - <expression> → <expression> + <expression>
  - <expression> → <expression> - <expression>
  - <expression> → <expression> * <expression>
  - <expression> → <expression> / <expression>

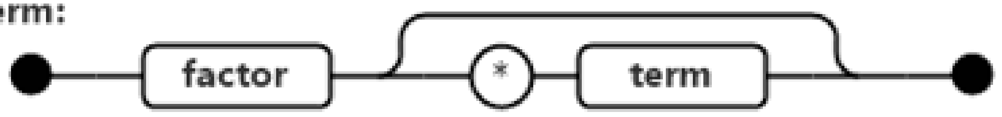
digit:



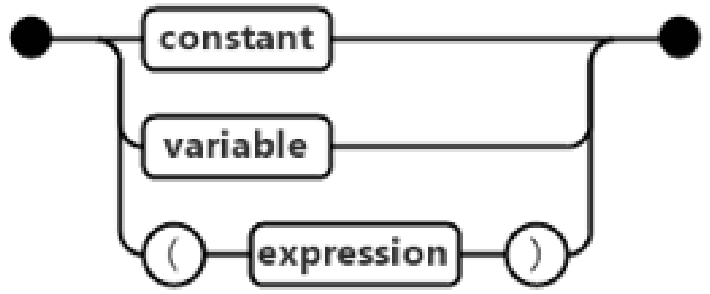
expression:



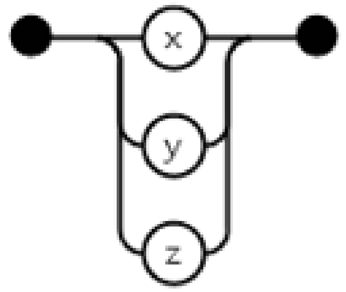
term:



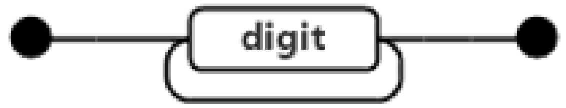
factor:



variable:



constant:



Programming Language and Programs

- Programs are coded in programming languages. Programming languages' grammar can be described in context-free grammar recursively.
- Therefore, all programs can be represented and defined recursively. They can all be evaluated recursively.

Repeated Pattern

LECTURE 2

Sequence

- A sequence [1, 2, 3, 4, 5, …, n] can be defined as

  $a_1 = 1$, **(Terminal Condition)**

  $a_2 = a_1+1$,

  $a_3 = a_2+1 = \mathbf{(a_1+1)}+1$, **(Repeated Pattern)**

  … ,

  $a_n = a_{n-1}+1$, (Recursive formula)

  When data or functions are defined by their own sub-cases, we call it recursive definition.

Recursion

- Base Condition (Termination Condition)
- Recursive Structure
  - Divide and Conquer: Split a big problem to smaller ones
  - Incremental Design: explore the relationship between small ones and the large one
  - Recursive Formula: describe the recursive relationship into Java code

Count Down Example

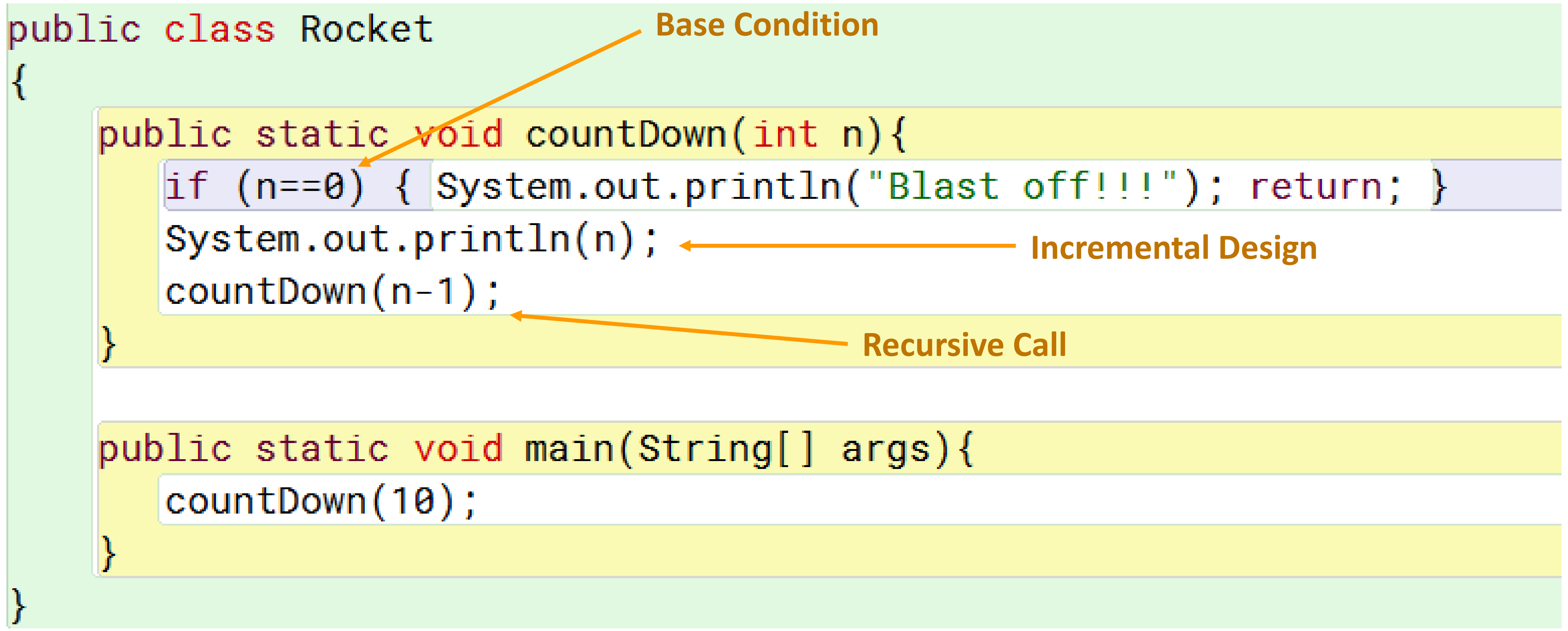
LECTURE 3

Countdown Recursion

- Call a recursive function with a counter.
- The counter decreases for each sub-case.
- When counter reaches 0, the rocket blasts off.

Demo Program:

Rocket.java



```
public class Rocket
{
    public static void countDown(int n){
        if (n==0) { System.out.println("Blast off!!!"); return; }
        System.out.println(n);
        countDown(n-1);
    }

    public static void main(String[] args){
        countDown(10);
    }
}
```

Demonstration Program

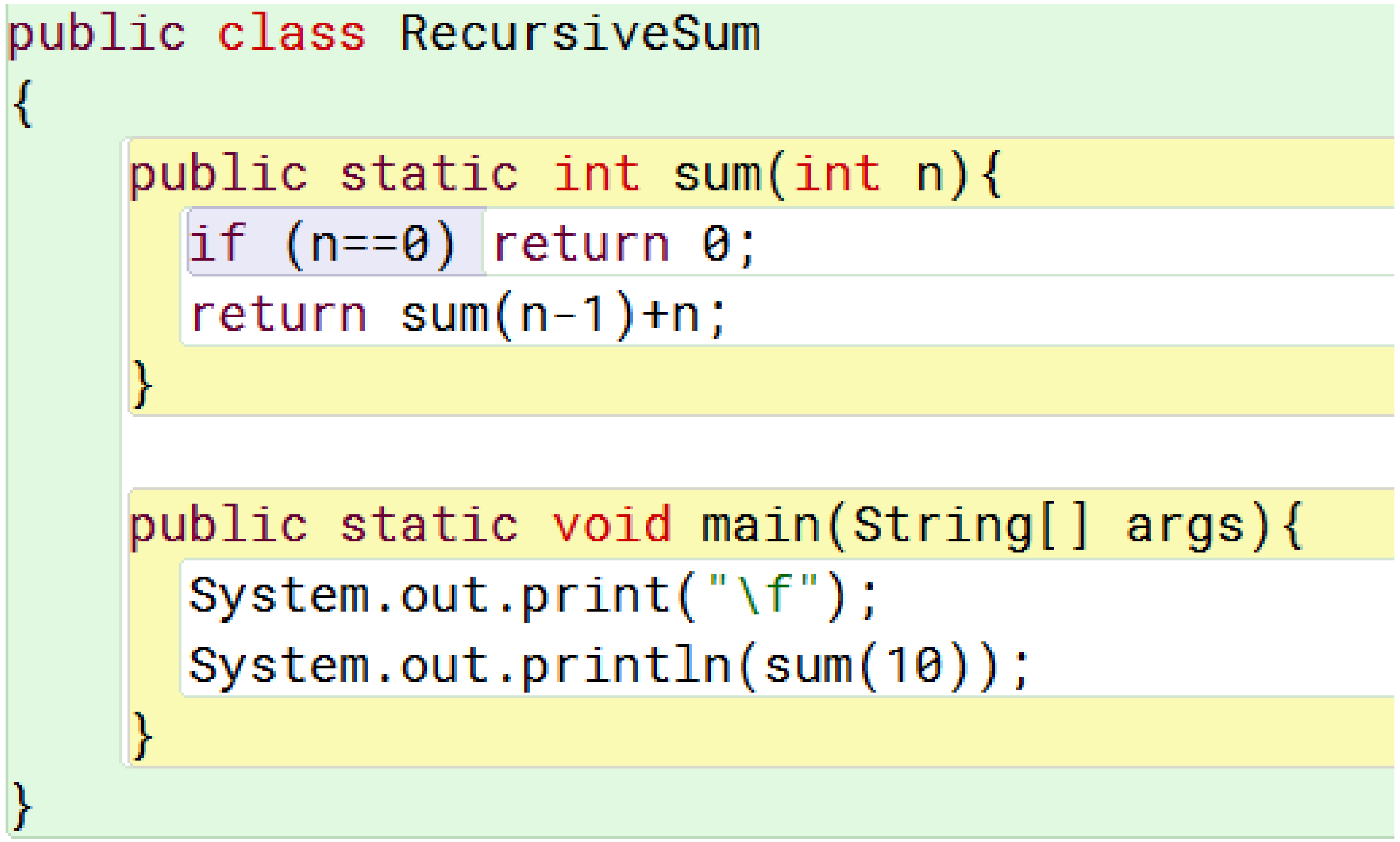
ROCKET.JAVA

Countdown Recursion with Return Value

- Call a recursive function with a counter.
- The counter decreases for each sub-case.
- When counter reaches 0, the base case return a value.
- Each bigger case receive a return value and returning another value.

Demo Program:

RecursiveSum.java



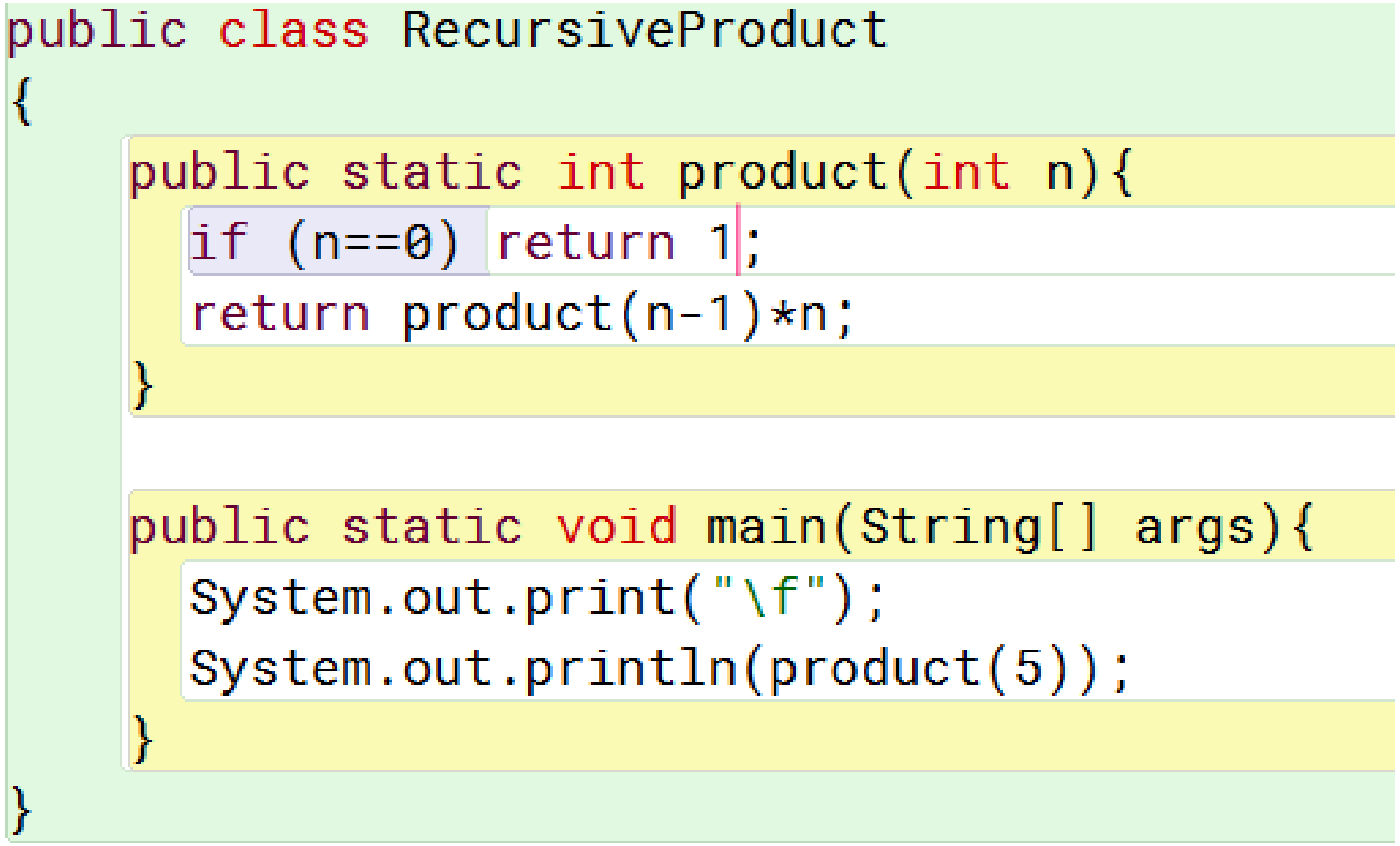
```
public class RecursiveSum
{
    public static int sum(int n){
      if (n==0) return 0;
      return sum(n-1)+n;
    }

    public static void main(String[] args){
      System.out.print("\f");
      System.out.println(sum(10));
    }
}
```

Demonstration Program

RECURSIVESUM.JAVA

Demo Program:

RecursiveProduct.java



```java
public class RecursiveProduct
{
    public static int product(int n){
      if (n==0) return 1;
      return product(n-1)*n;
    }

    public static void main(String[] args){
      System.out.print("\f");
      System.out.println(product(5));
    }
}
```

Demonstration Program

RECURSIVEPRODUCT.JAVA

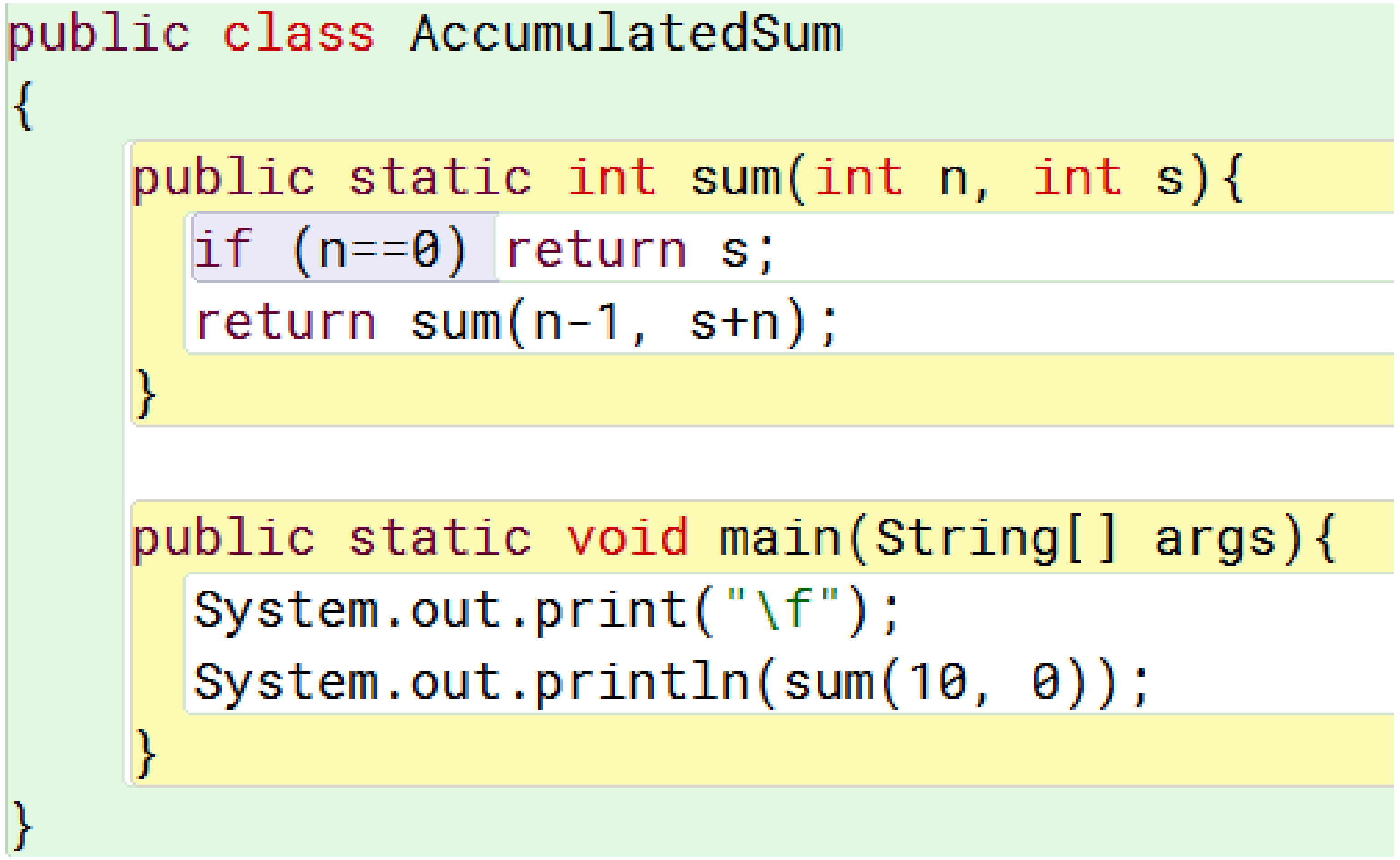
Recursion with Accumulator

- Call a recursive function with an accumulator as a parameter.
- In the base case, the accumulator is returned.
- Each recursive call, the accumulator is updated (accumulates).

Demonstration Program

ACCUMULATEDSUM.JAVA

Demo Program:
AccumulatedSum.java



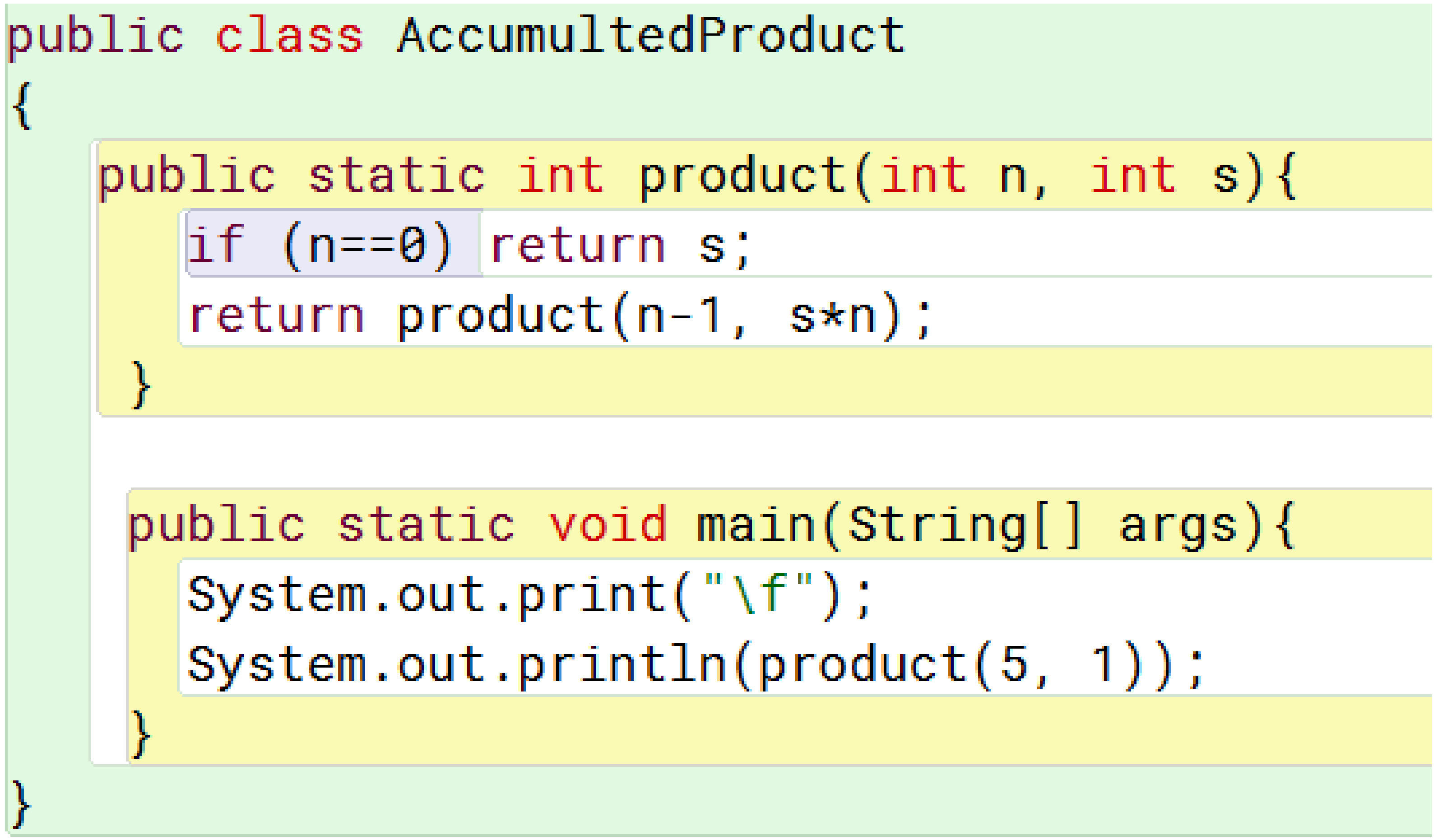
```java
public class AccumulatedSum
{
    public static int sum(int n, int s){
      if (n==0) return s;
      return sum(n-1, s+n);
    }

    public static void main(String[] args){
      System.out.print("\f");
      System.out.println(sum(10, 0));
    }
}
```

Demonstration Program

ACCUMULATEDPRODUCT.JAVA

Demo Program:

AccumulatedProduct.java



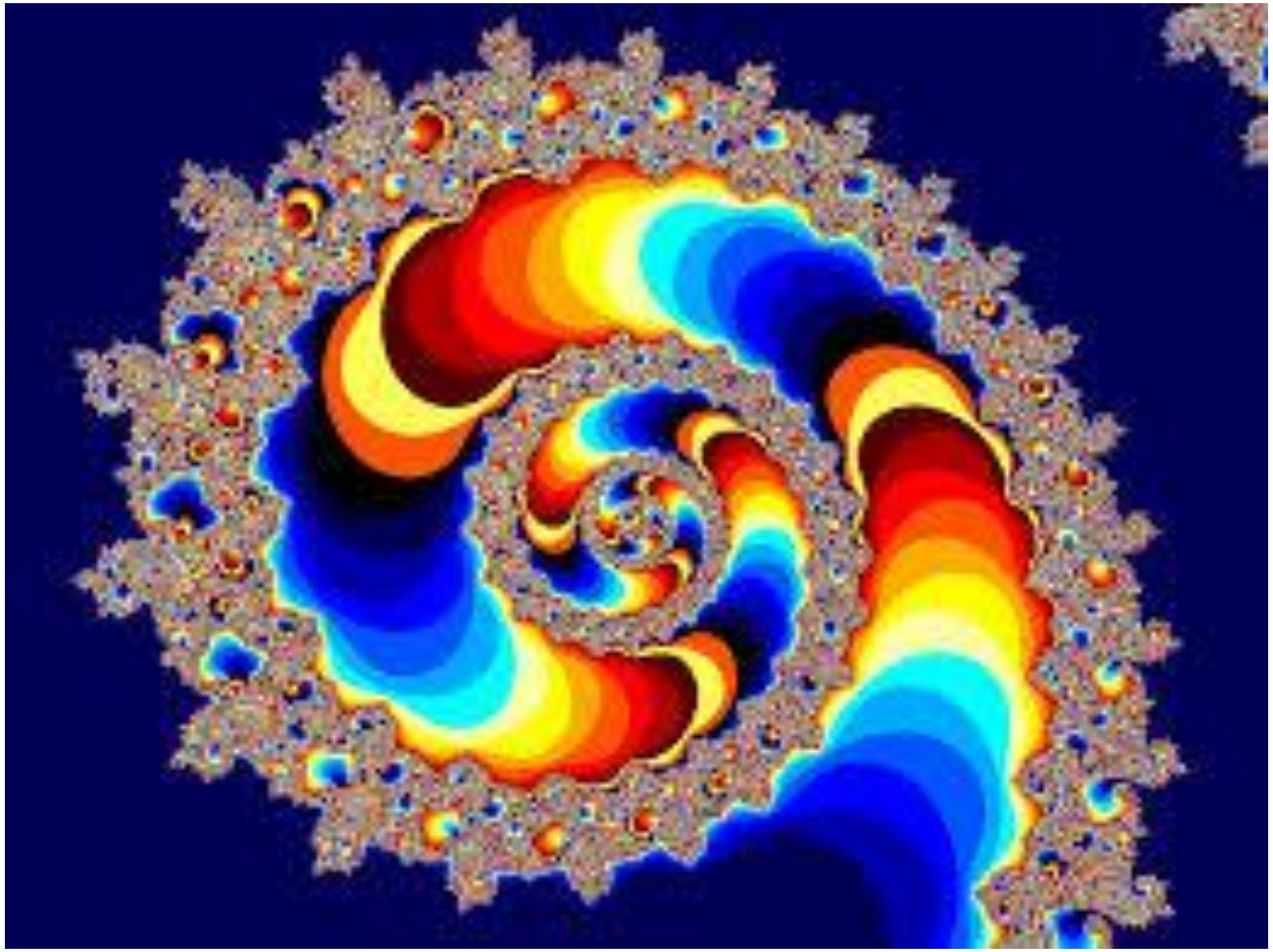
```
public class AccumultedProduct
{
    public static int product(int n, int s){
        if (n==0) return s;
        return product(n-1, s*n);
    }

    public static void main(String[] args){
        System.out.print("\f");
        System.out.println(product(5, 1));
    }
}
```

Why with accumulator?

- Tail recursion.
- More efficient than normal recursive function.
- As a recursive helper function.

Fractals

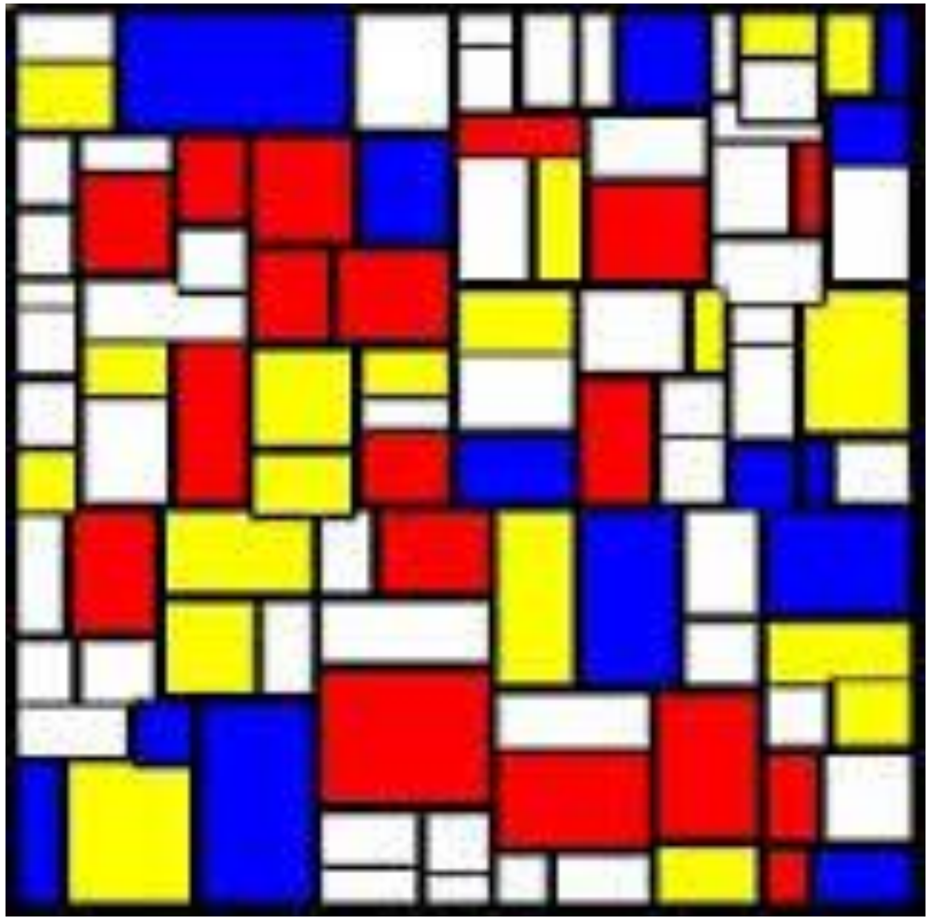
LECTURE 4

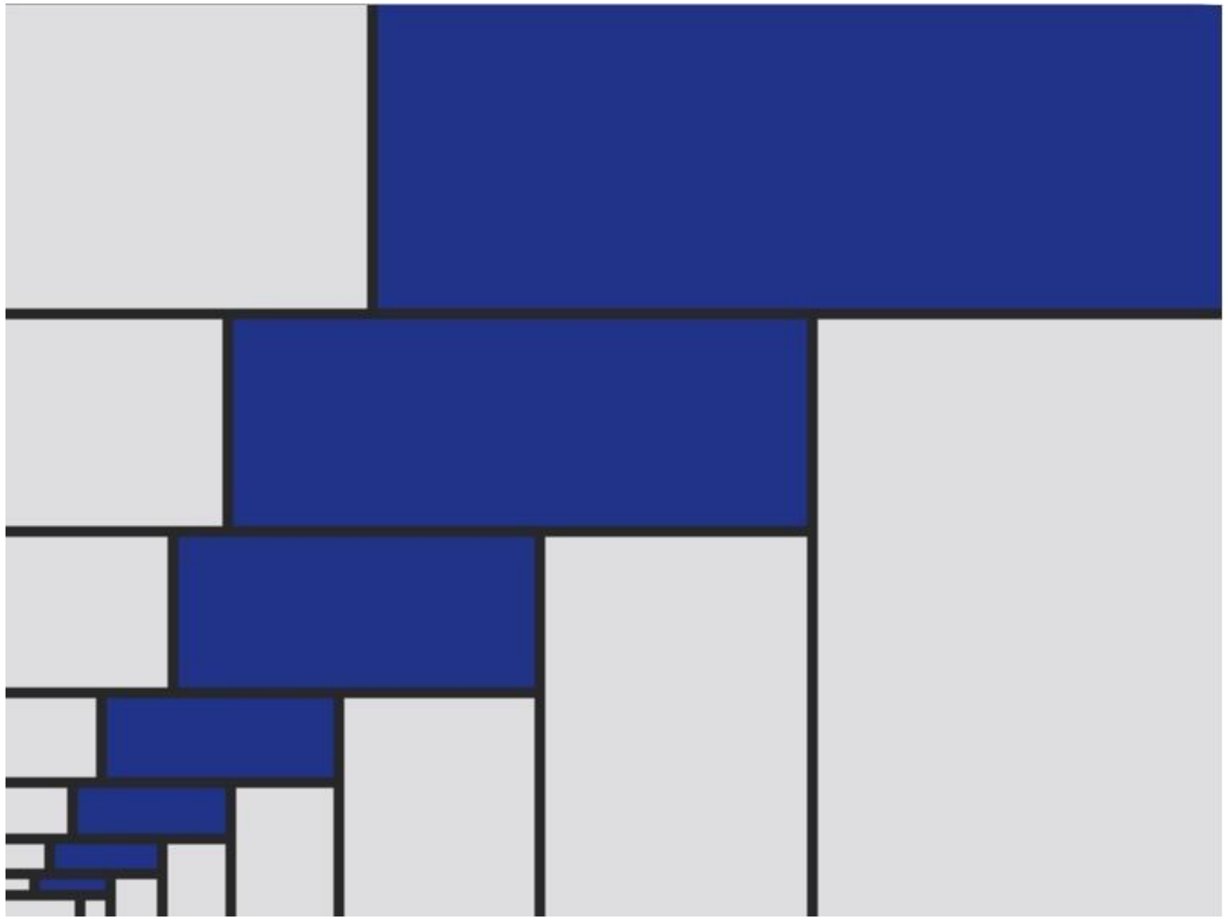


Fractals (Computer Art)

Mondrimat Arts

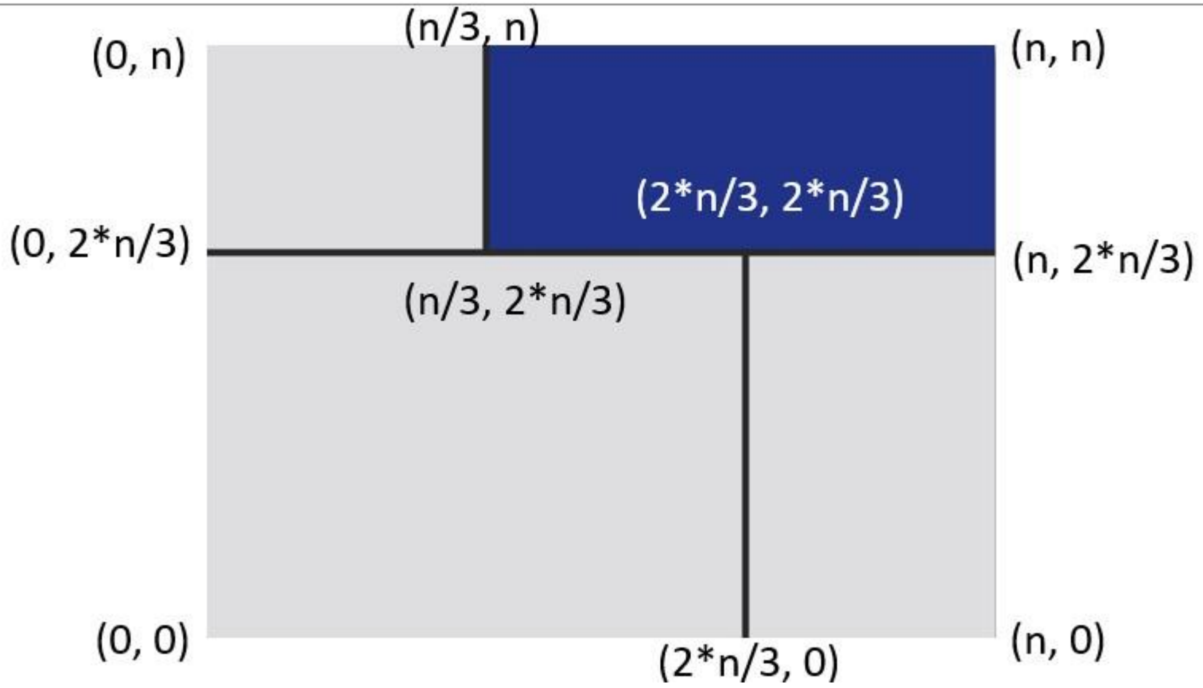
http://www.stephen.com/mondrimat/



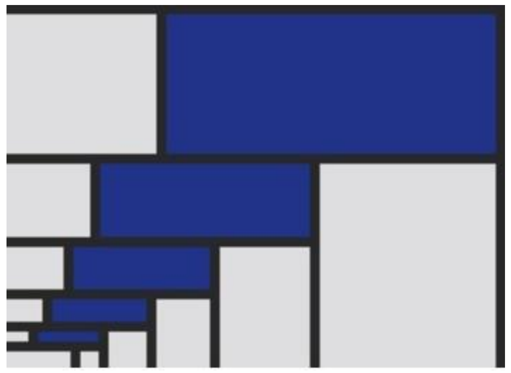
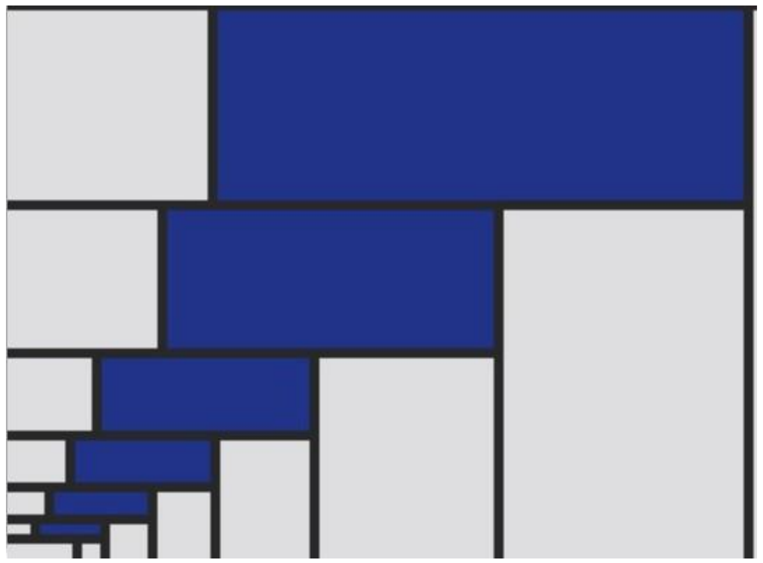
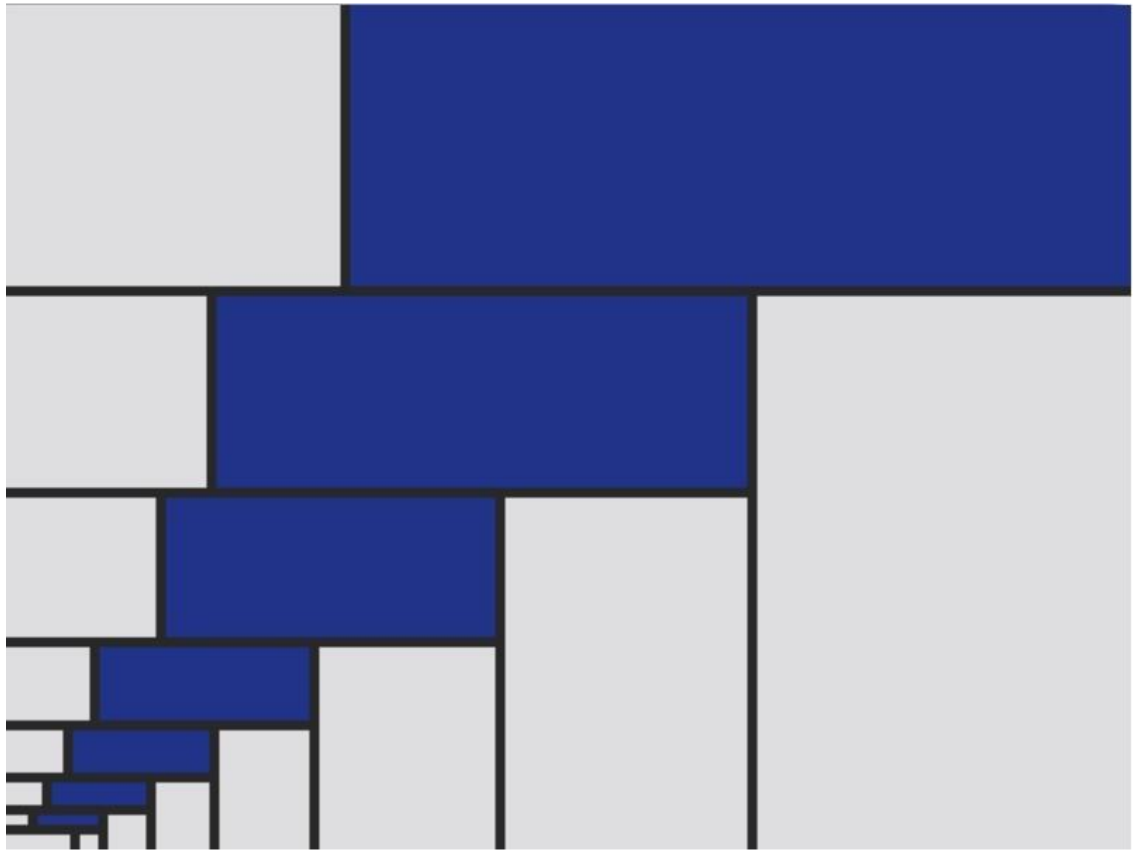


Reading the Recursive Pattern

draw(int n) { /* will draw the following pattern for one recursion call. */ }



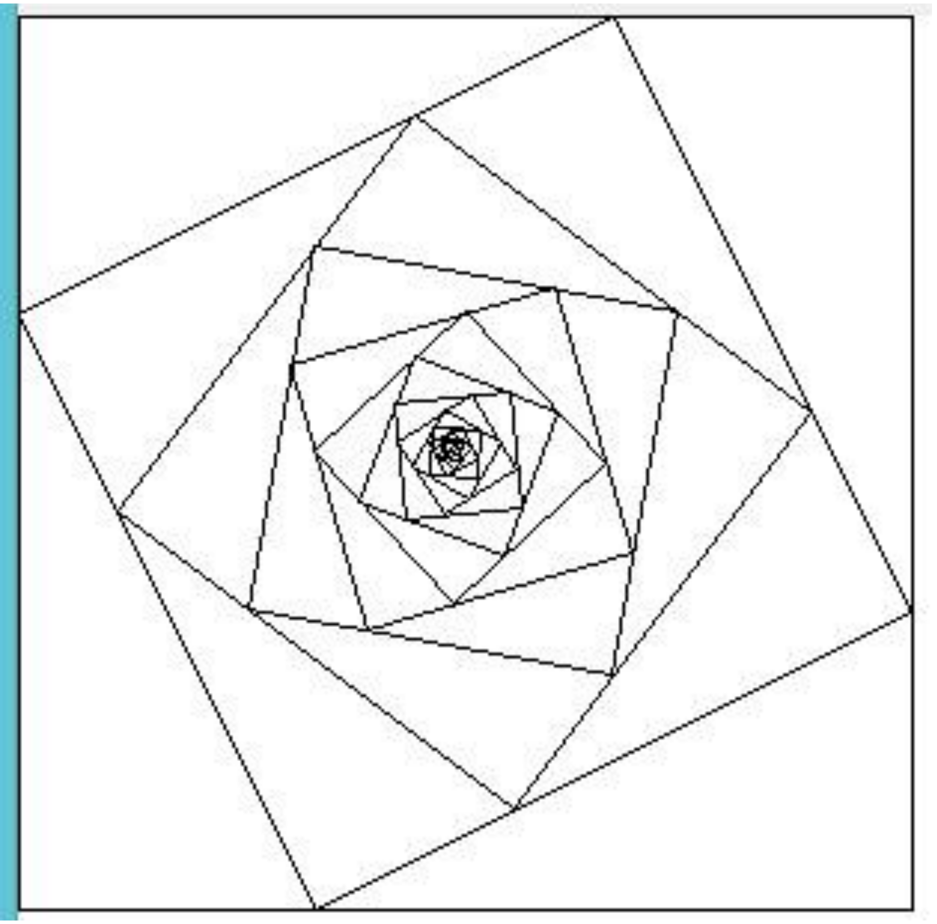
Reading the Recursive Pattern

Three Principles in Recursion

1. Divide and Conquer
2. Recursive Function
3. Stop Condition (Otherwise, it will never stop.)

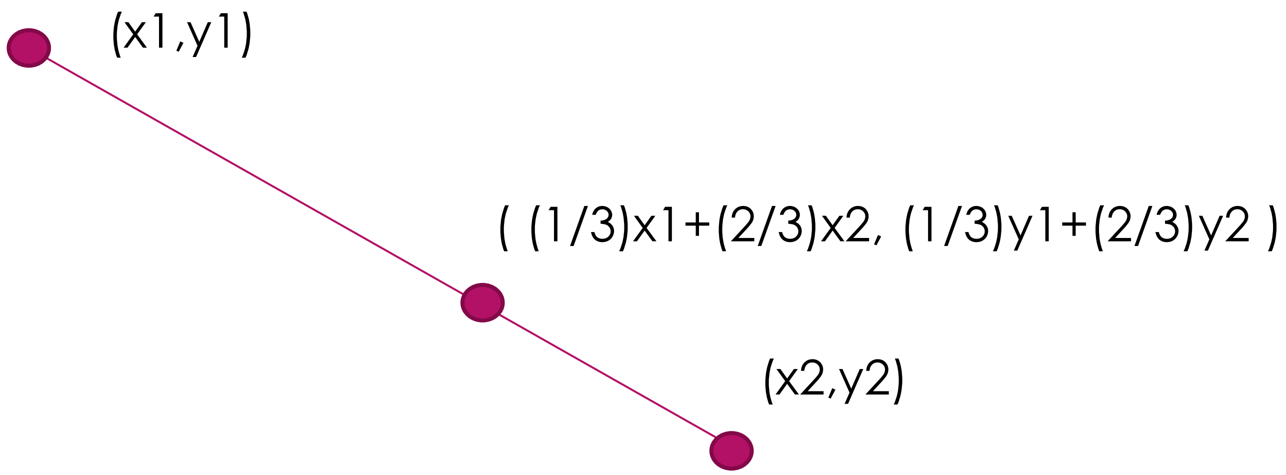
Demo Program: RecursiveRectangle.java



- This recursive rectangle project requires you to draw rectangles recursively on the one-third points of the line segments as shown on the left.
- After finishing a rectangle, you need to draw smaller rectangles based on the one-third points of the previous rectangle's line segments. (Recursion)
- Until the line segment's length is less than square root of ten, which also means the square of the line segment's length is less than 10. Then, you stop the recursion. (Stop Condition)
- Your results should look like the picture on the left side.

Demo Program: RecursiveRectangle.java

Calculation of the one-third point



Demonstration Program

RECURSIVERECTANGLE.JAVA

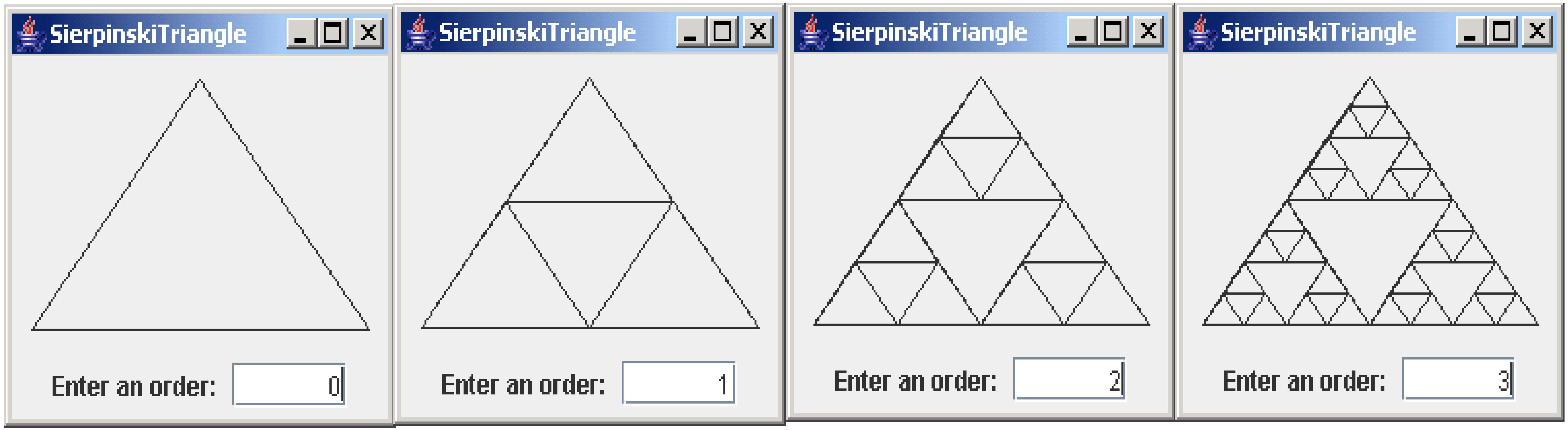
Fractals?

- A fractal is a geometrical figure just like triangles, circles, and rectangles, but fractals can be divided into parts, each of which is a reduced-size copy of the whole.
- There are many interesting examples of fractals.
- This section introduces a simple fractal, called *Sierpinski triangle*, named after a famous Polish mathematician.

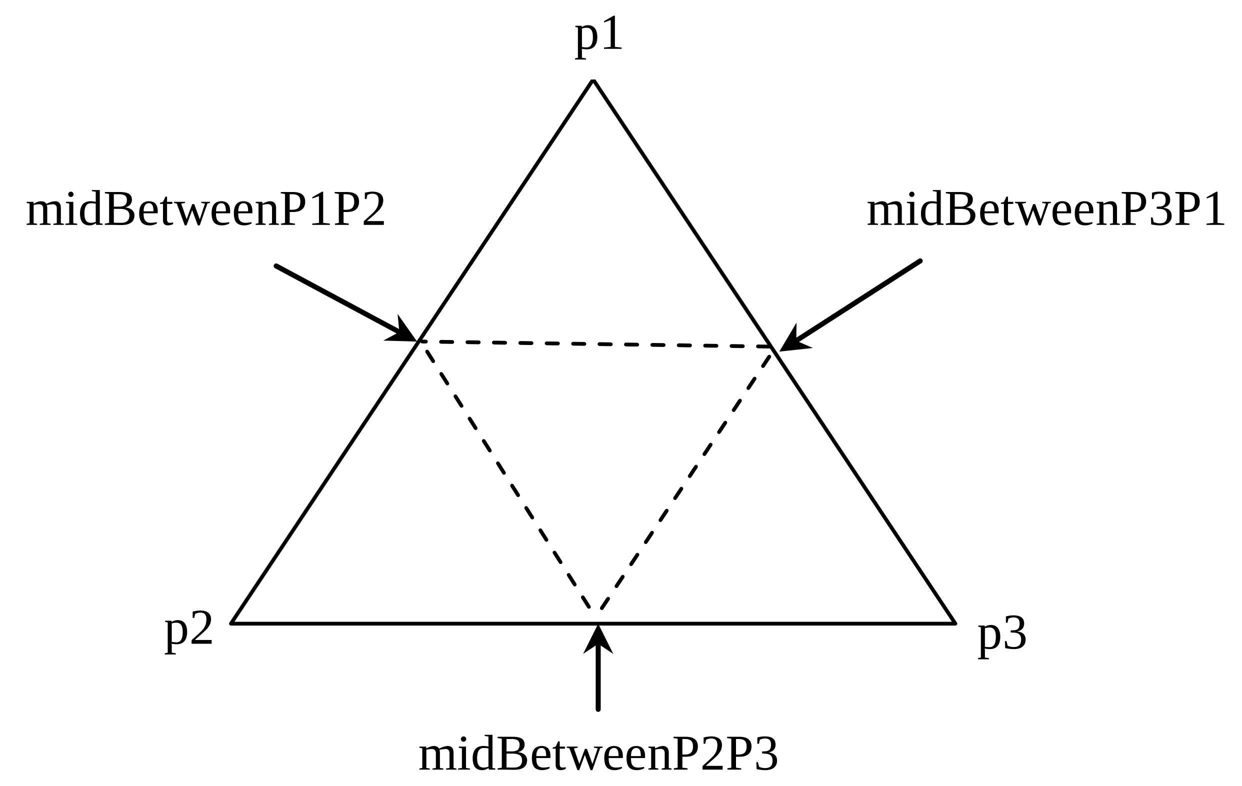
Sierpinski Triangle

- It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
- Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
- Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2, as shown in Figure (c).
- You can repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on, as shown in Figure (d).

Recursion

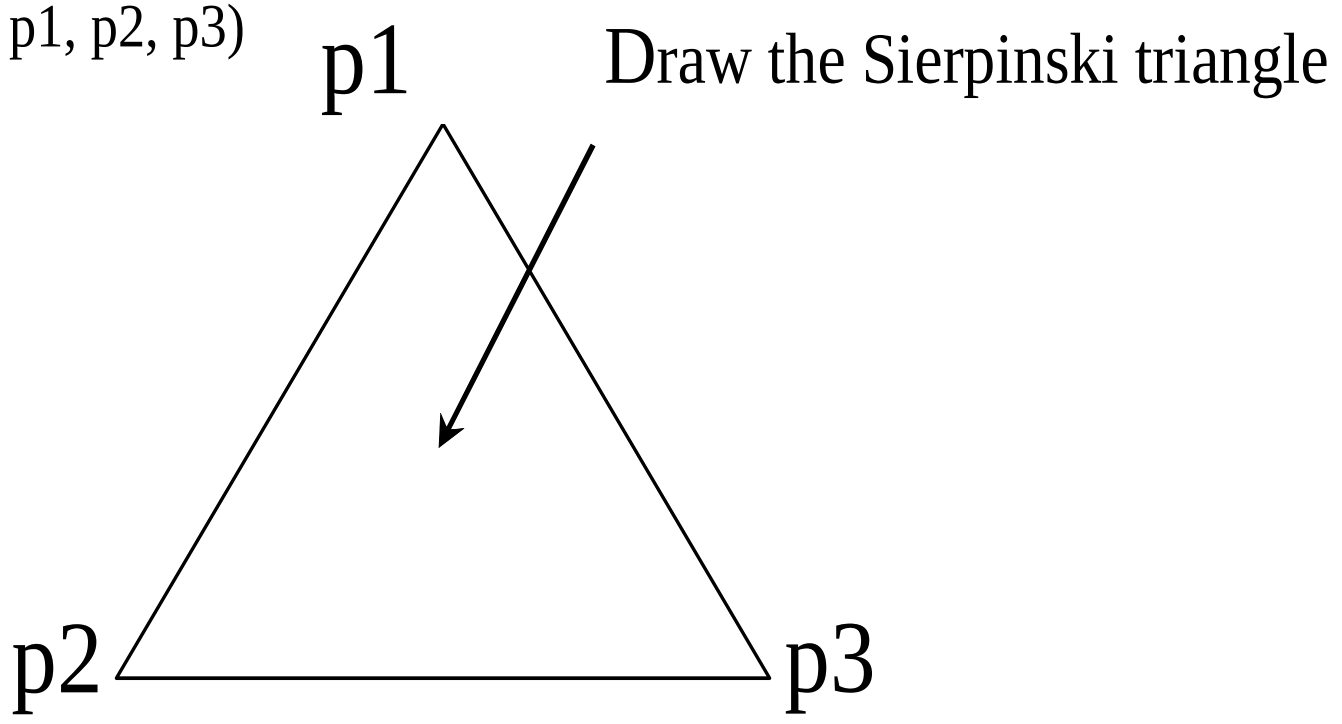

SierpinskiTriangle
Enter an order: 0
SierpinskiTriangle
Enter an order: 1
SierpinskiTriangle
Enter an order: 2
SierpinskiTriangle
Enter an order: 3

Sierpinski Triangle Solution

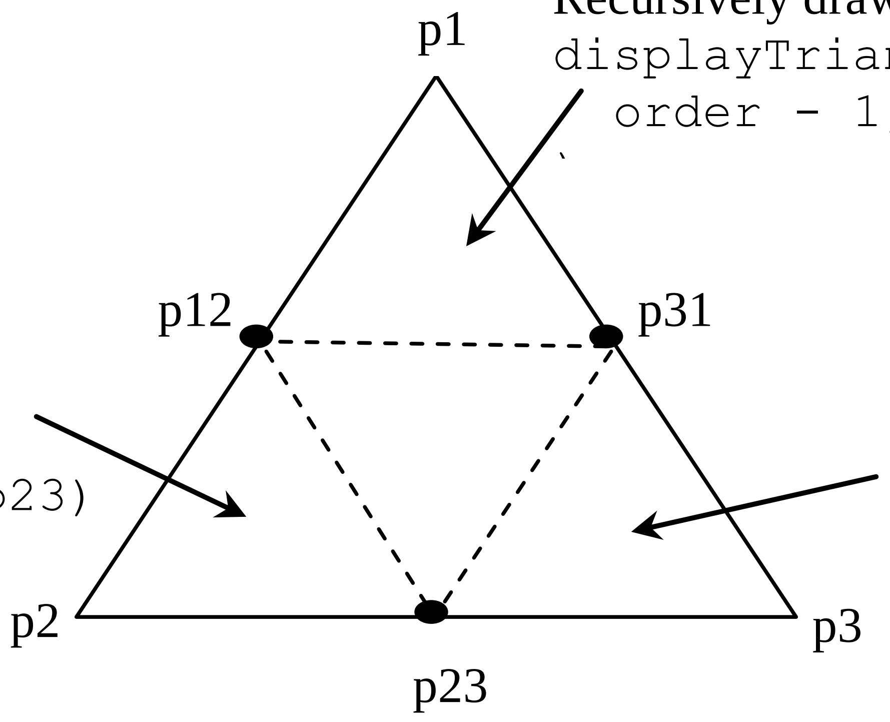


Sierpinski Triangle Solution

displayTriangles(g, order, p1, p2, p3)



Sierpinski Triangle Solution



Recursively draw the small Sierpinski triangle

```
displayTriangles(g,
  order - 1, p1, p12, p31)
```

Recursively draw the small Sierpinski triangle

```
displayTriangles(g,
  order - 1, p12, p2, p23)
```

Recursively draw the small Sierpinski triangle

```
displayTriangles(g,
  order - 1, p31, p23, p3)
```

Demonstration Program

SIERPINSKITRIANGLES.JAVA

Tail Recursion

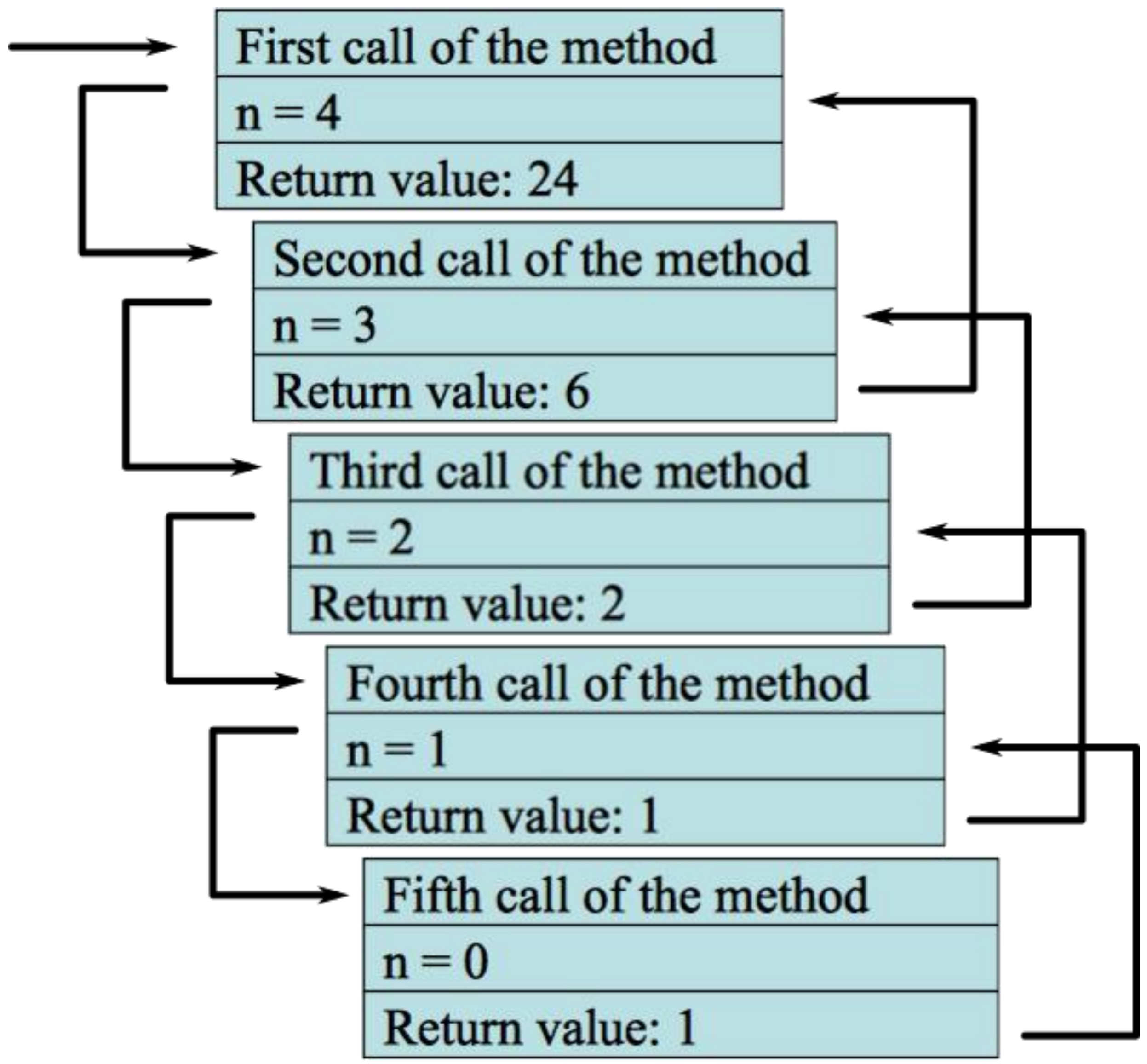
LECTURE 5

Tail Recursion

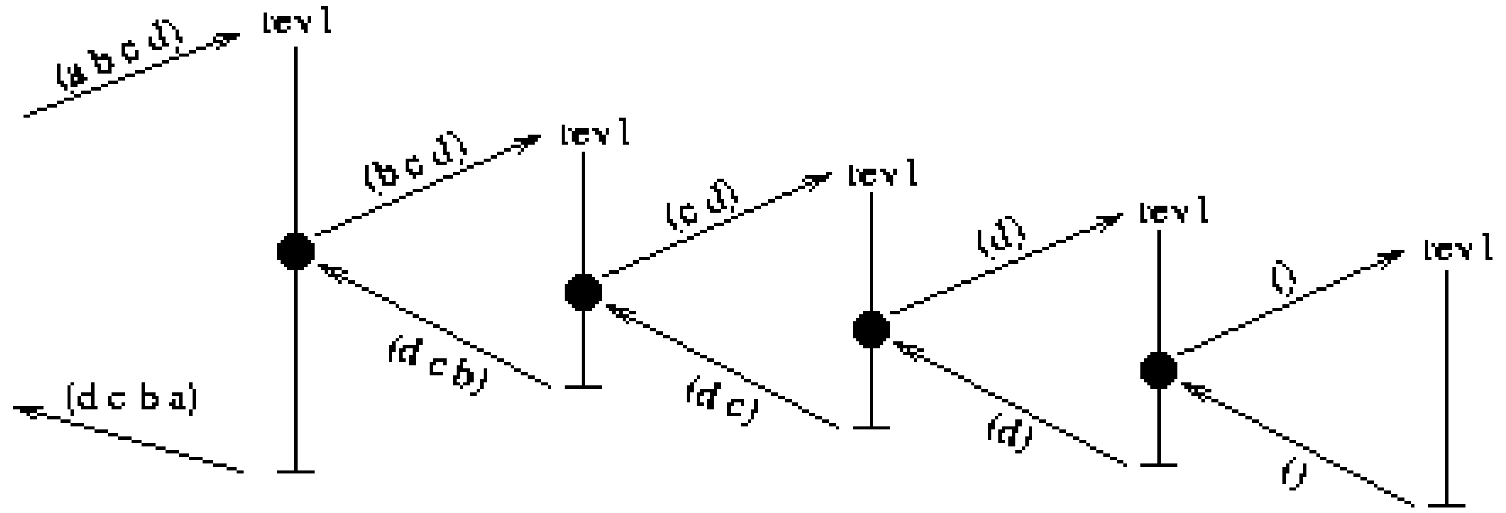
- A **tail recursion** is a **recursive function** where the **function** calls itself at the end ("**tail**") of the **function** in which no computation is done after the return of **recursive** call.
- Many compilers optimize to change a **recursive** call to a **tail recursive** or an iterative call. ...
- Hence the compiler can do away with a stack.

Tail Recursion

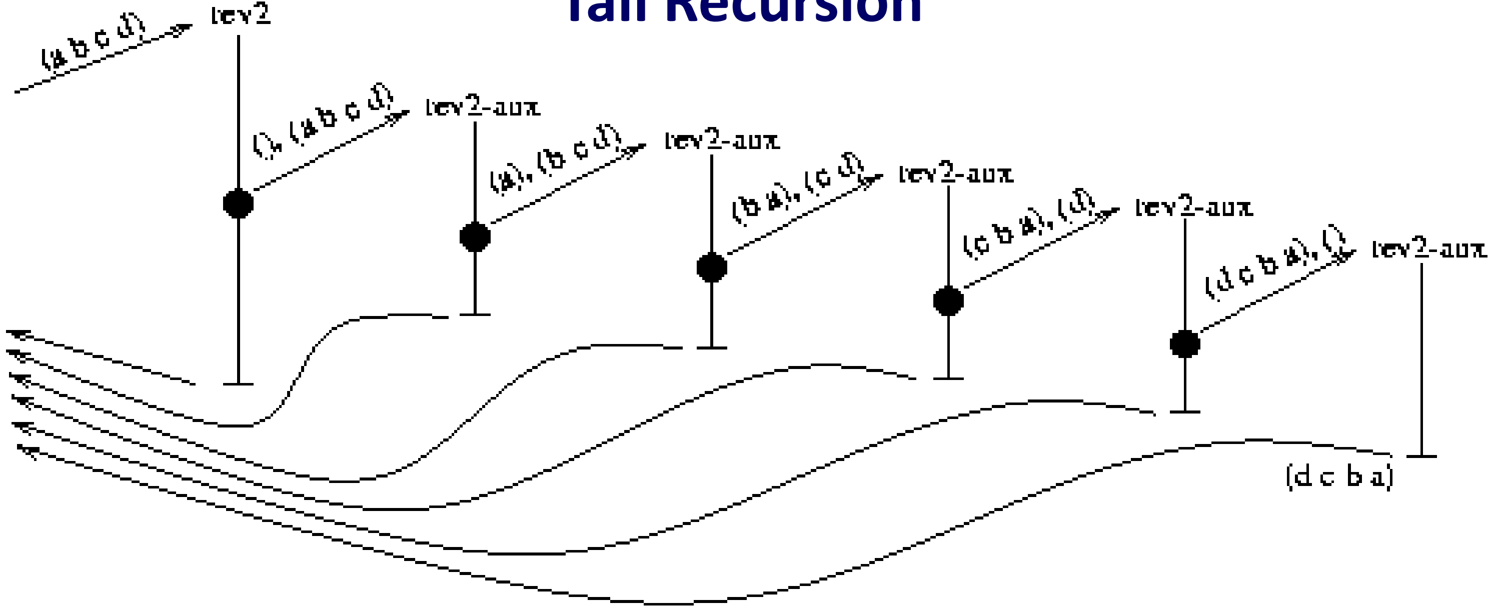
- A tail-recursive function is just a function whose very the last action is a call to itself. Tail-Call Optimisation (TCO) lets us convert regular recursive calls into tail calls to make recursions practical for large inputs, which was earlier leading to stack overflow error in normal recursion scenario.
- Java does not directly support TCO at the compiler level, but with the introduction of lambda expressions and functional interfaces in JAVA 8, we can implement this concept in a few lines of code.


First call of the method
n = 4
Return value: 24
Second call of the method
n = 3
Return value: 6
Third call of the method
n = 2
Return value: 2
Fourth call of the method
n = 1
Return value: 1
Fifth call of the method
n = 0
Return value: 1

Non-Tail Recursion



Tail Recursion



Computing Factorial

Example of Tail Recursion

LECTURE 6

Computing Factorial

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

n! = n * (n-1)!
```

Computing Factorial

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

```
factorial(3)
```

Computing Factorial

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

```
factorial(3) = 3 * factorial(2)
```

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

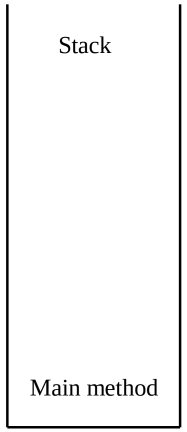
```
factorial(4) = 4 * factorial(3)
             = 4 * 3 * factorial(2)
             = 4 * 3 * (2 * factorial(1))
             = 4 * 3 * ( 2 * (1 * factorial(0)))
             = 4 * 3 * ( 2 * ( 1 * 1)))
             = 4 * 3 * ( 2 * 1)
             = 4 * 3 * 2
             = 4 * 6
             = 24
```

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

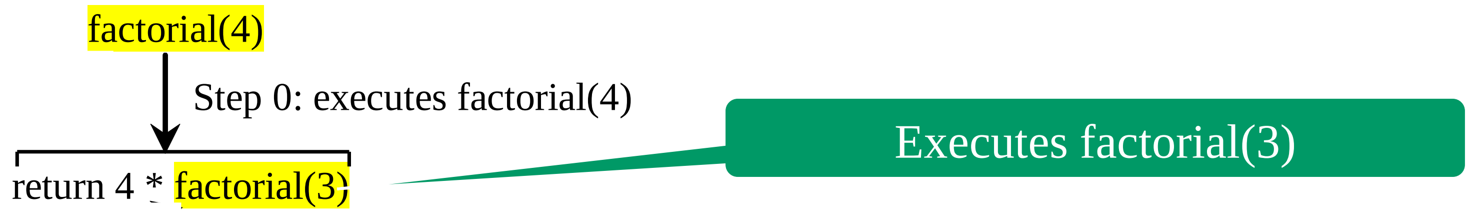
Trace Recursive factorial

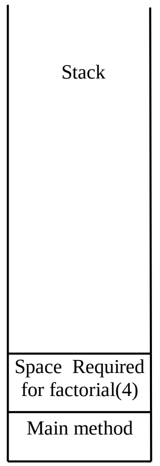
Executes factorial(4)

factorial(4)

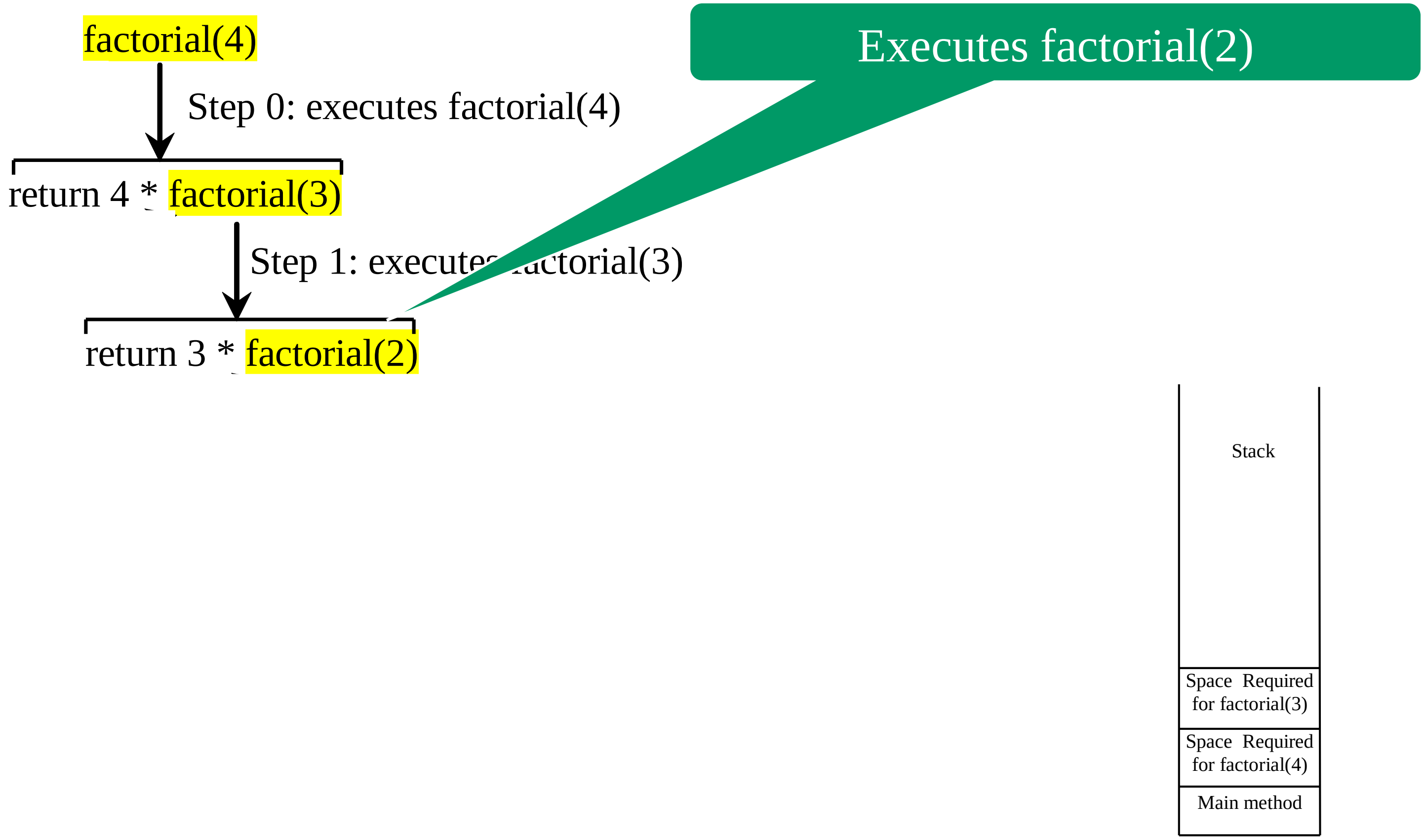


Trace Recursive factorial

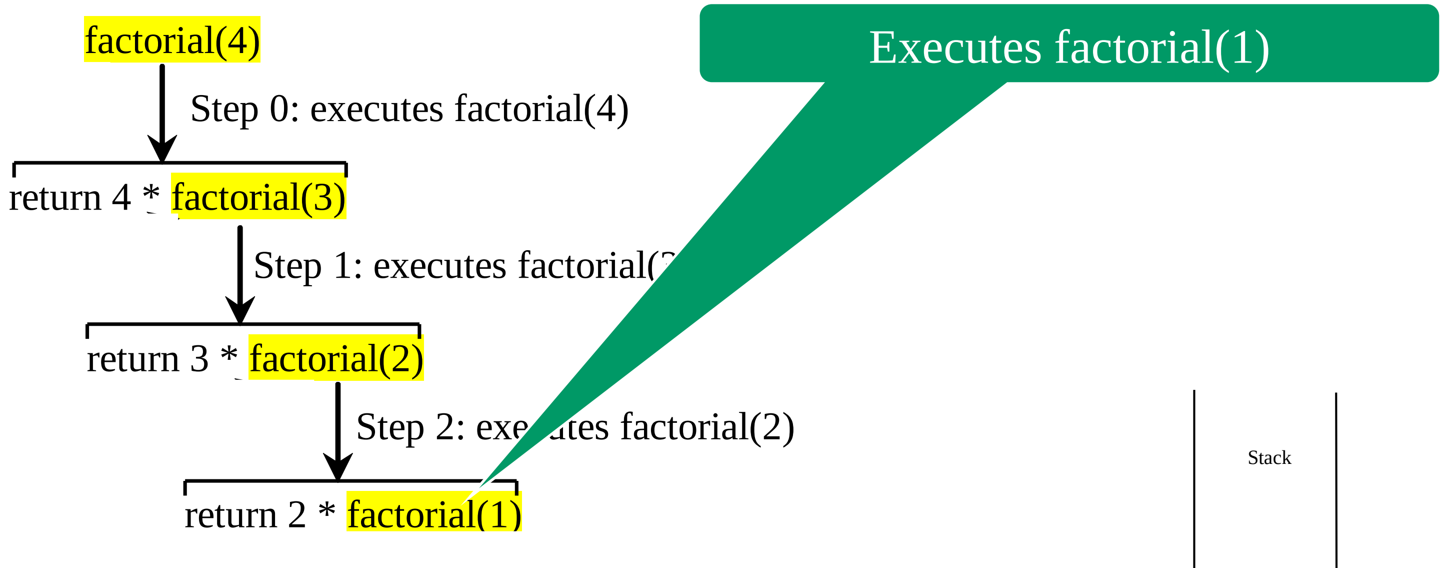


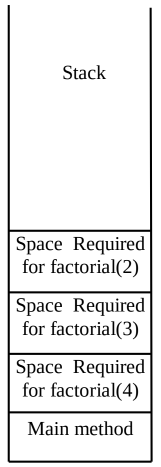


Trace Recursive factorial

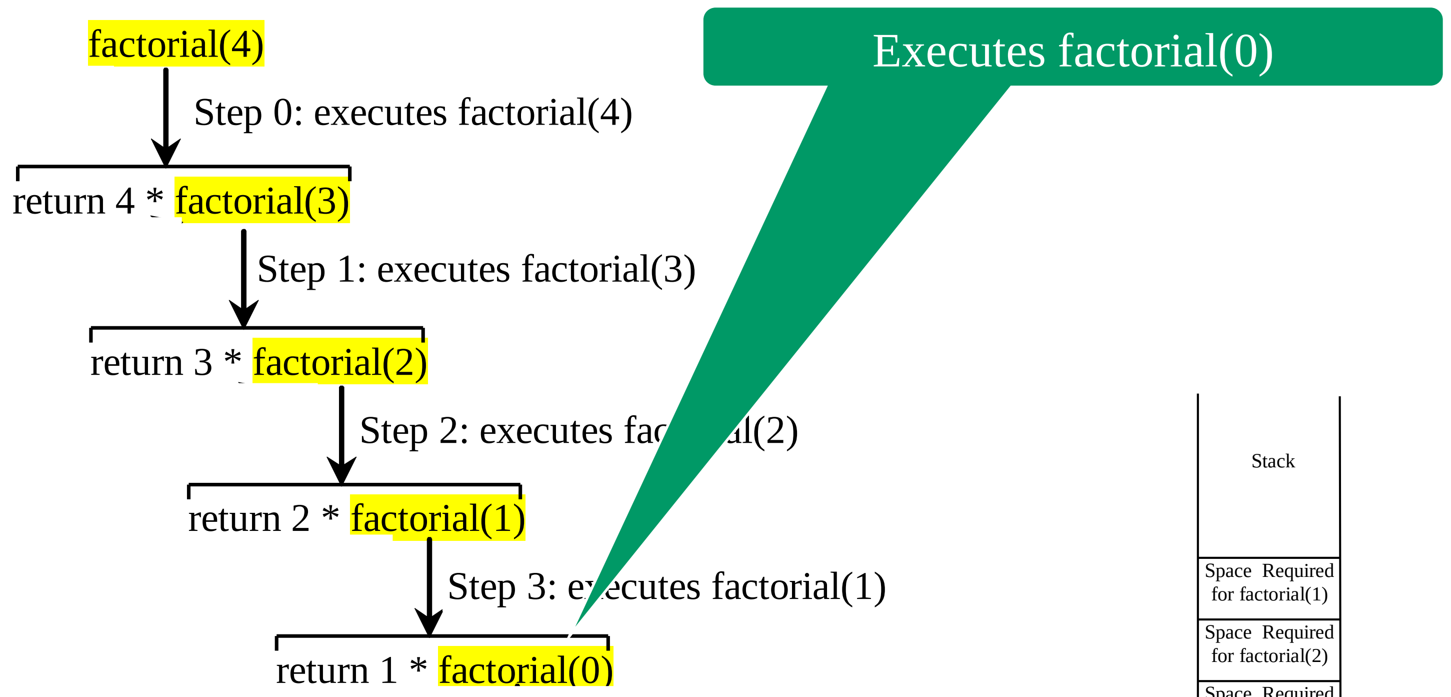

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Executes factorial(2)
Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

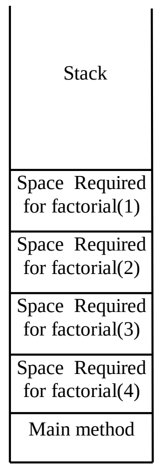
Trace Recursive factorial



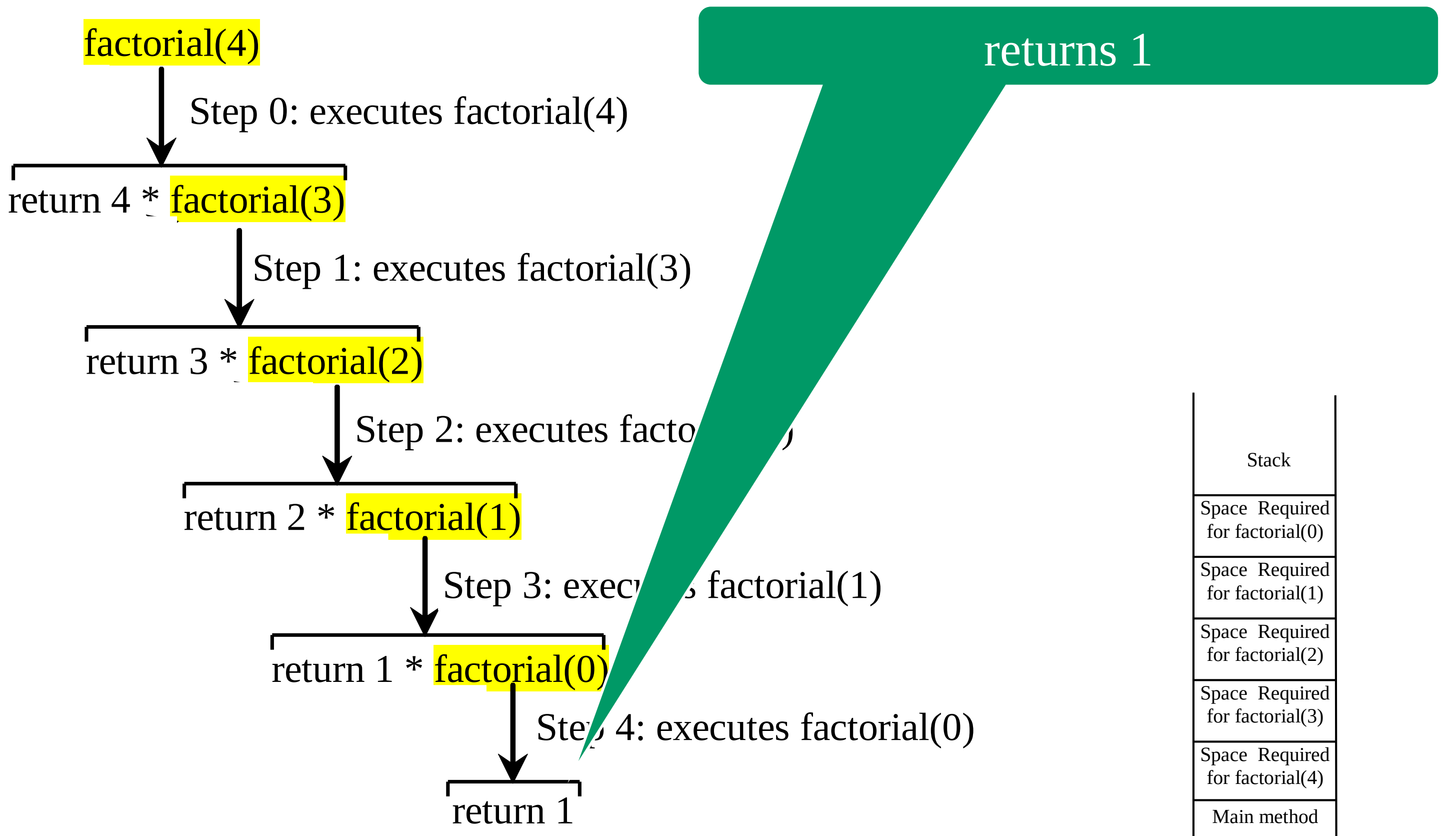

Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

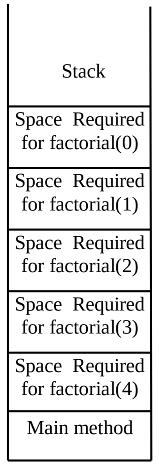
Trace Recursive factorial



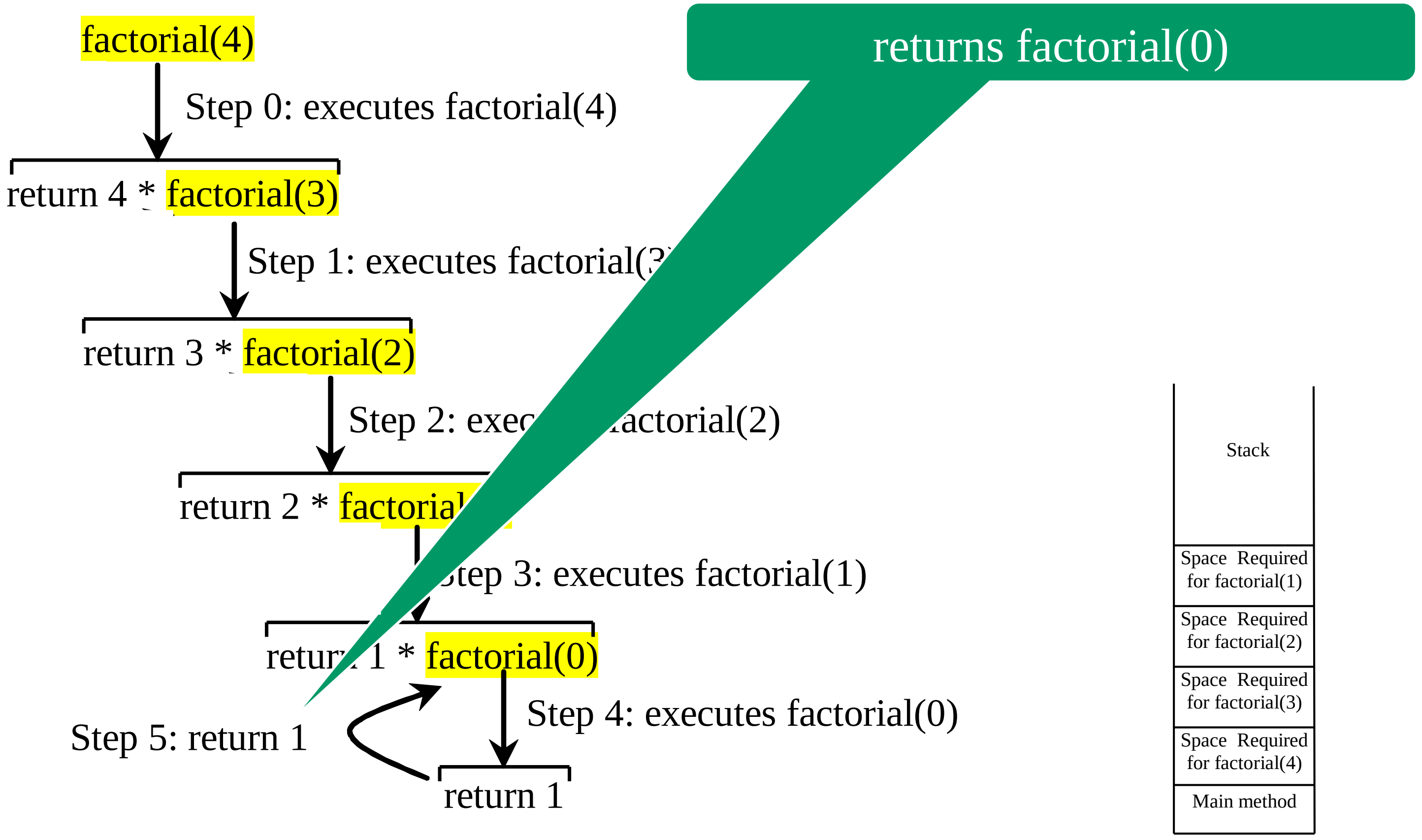

Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

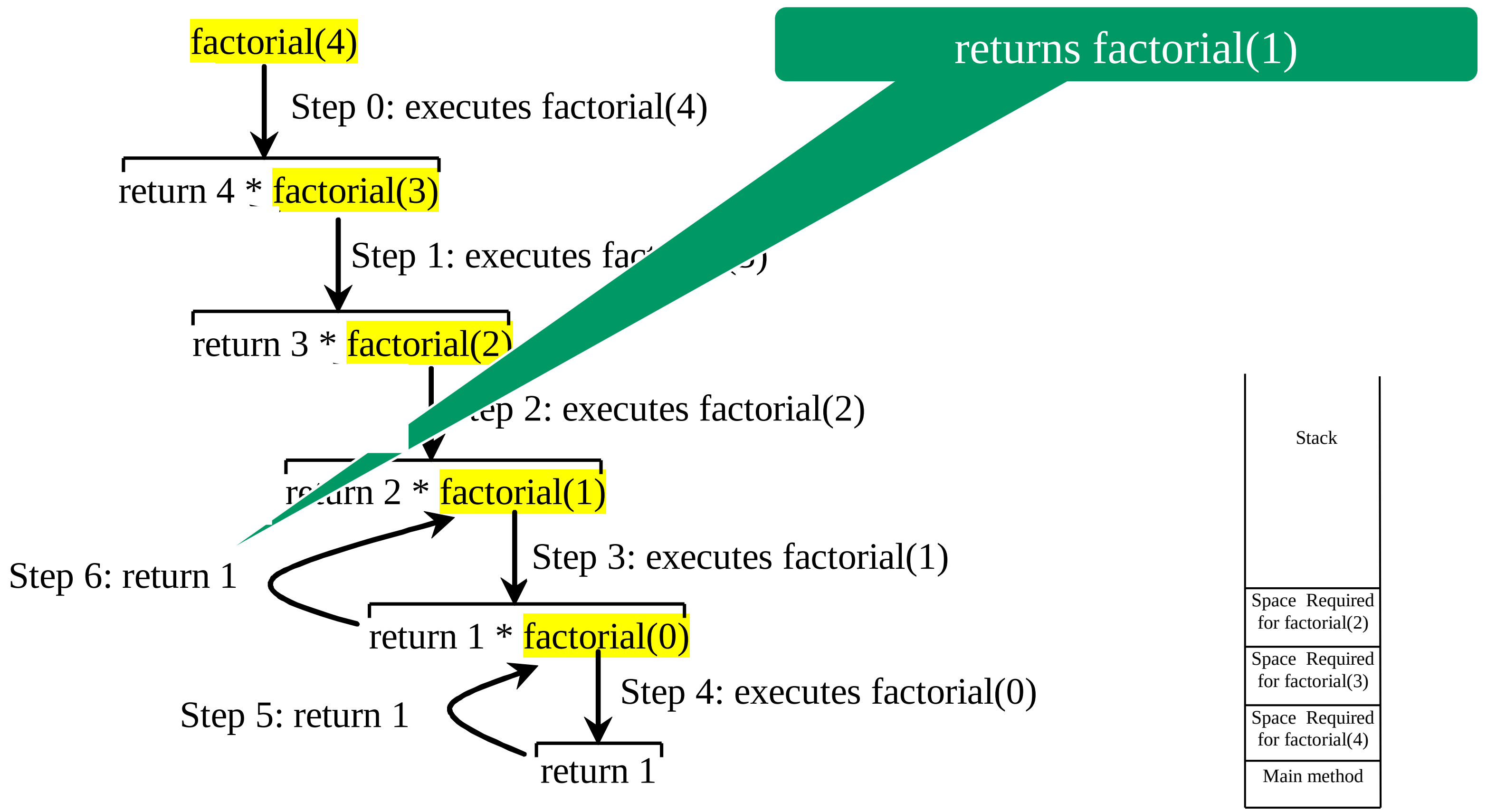



Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

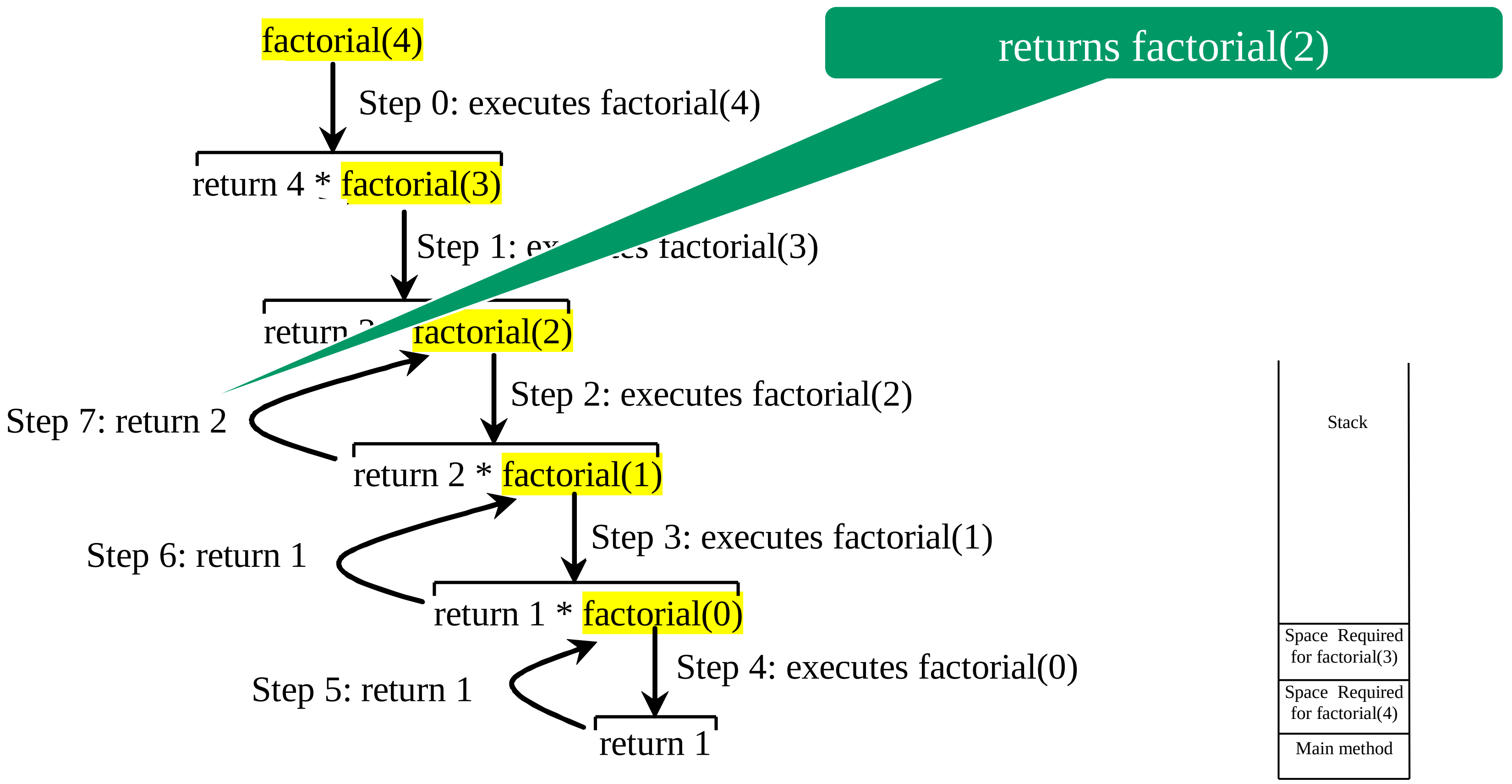

returns factorial(0)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

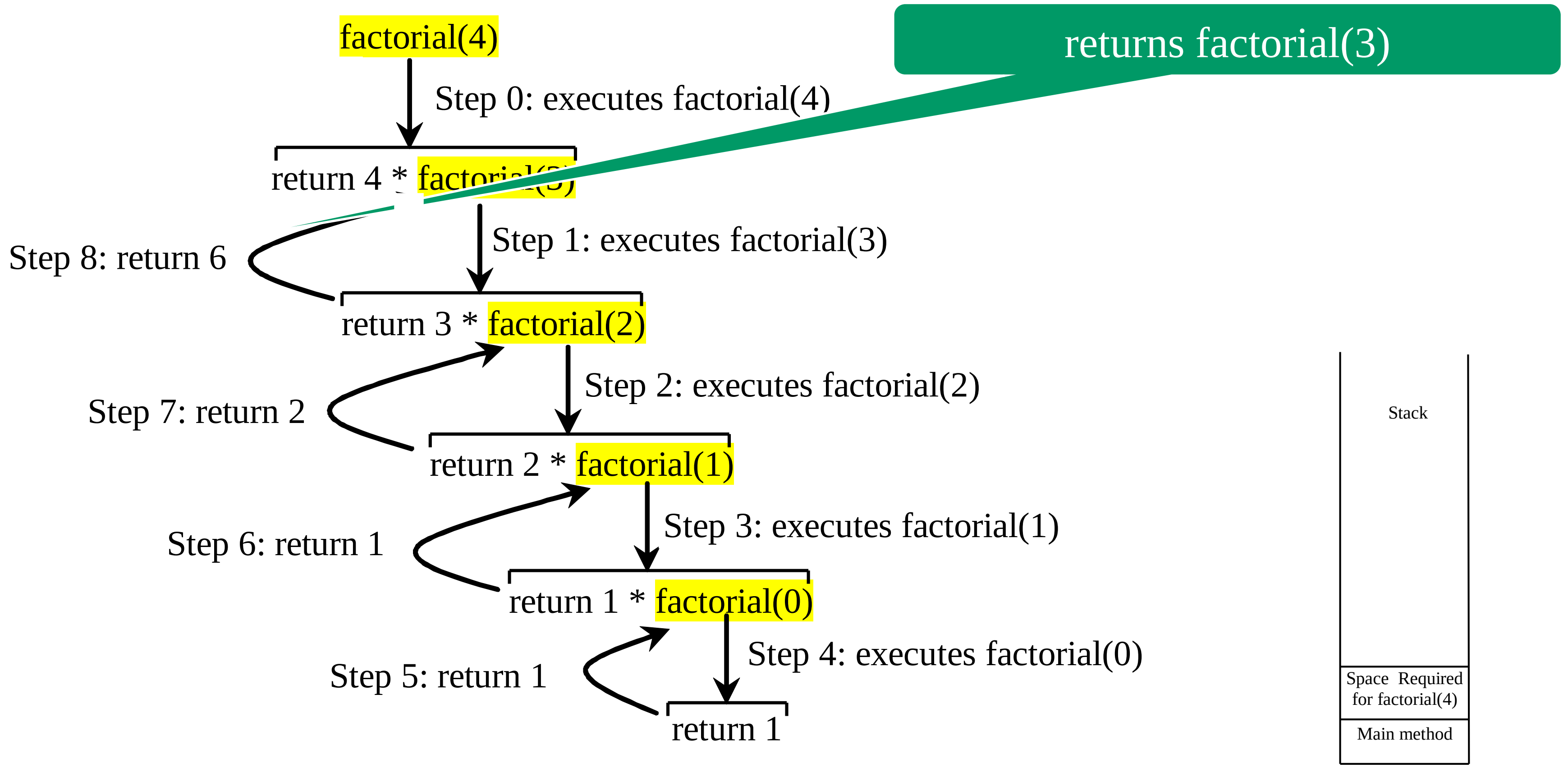


returns factorial(1)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes fac
return 3 * factorial(2)
ep 2: executes factorial(2)
re urn 2 * factorial(1)
Step 6: return 1
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 5: return 1
Step 4: executes factorial(0)
return 1
Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

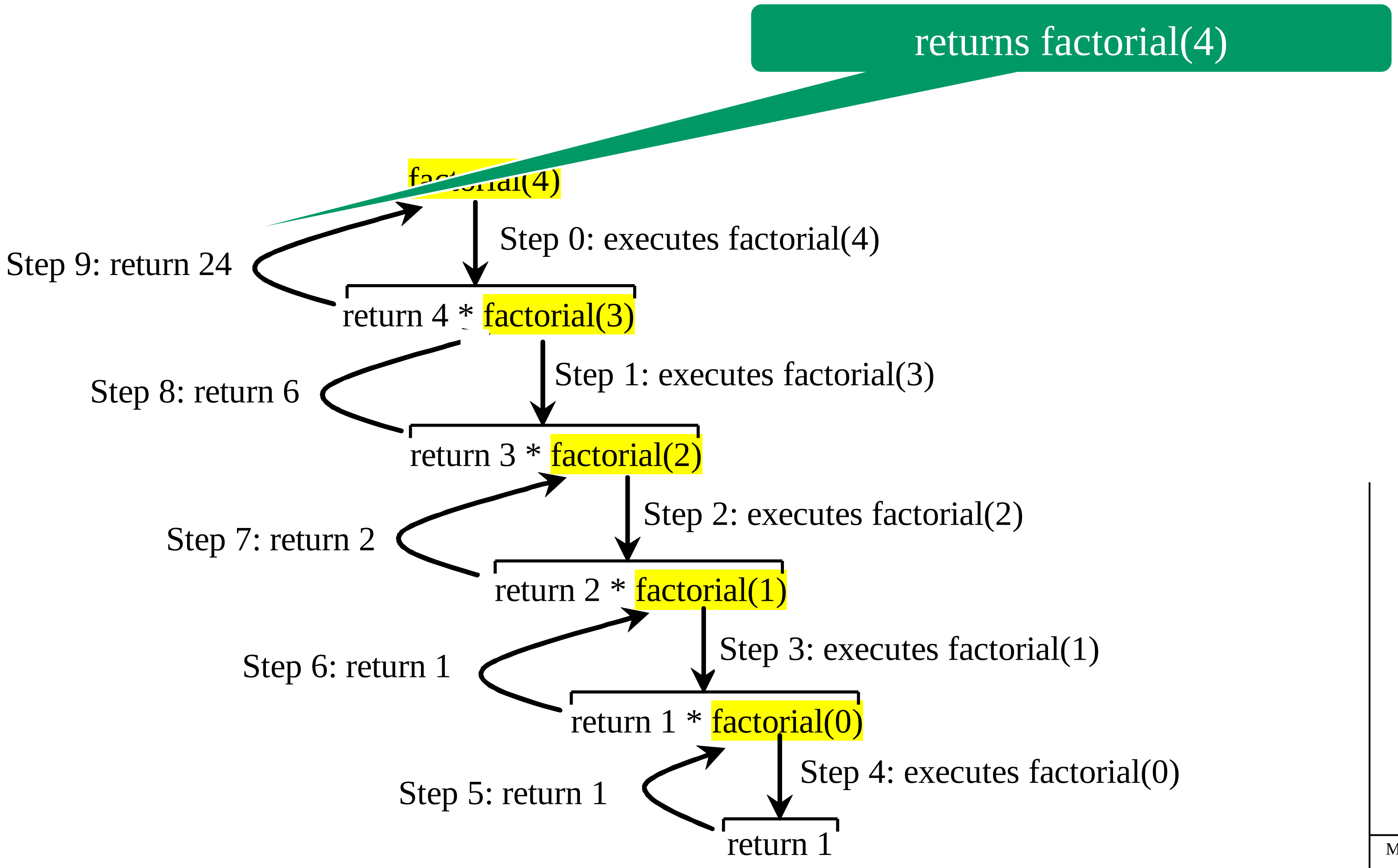
Trace Recursive factorial

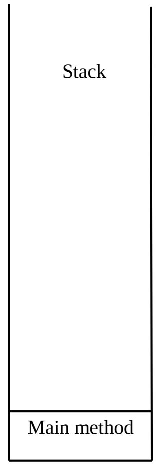

returns factorial(2)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Step 6: return 1
Step 7: return 2
Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

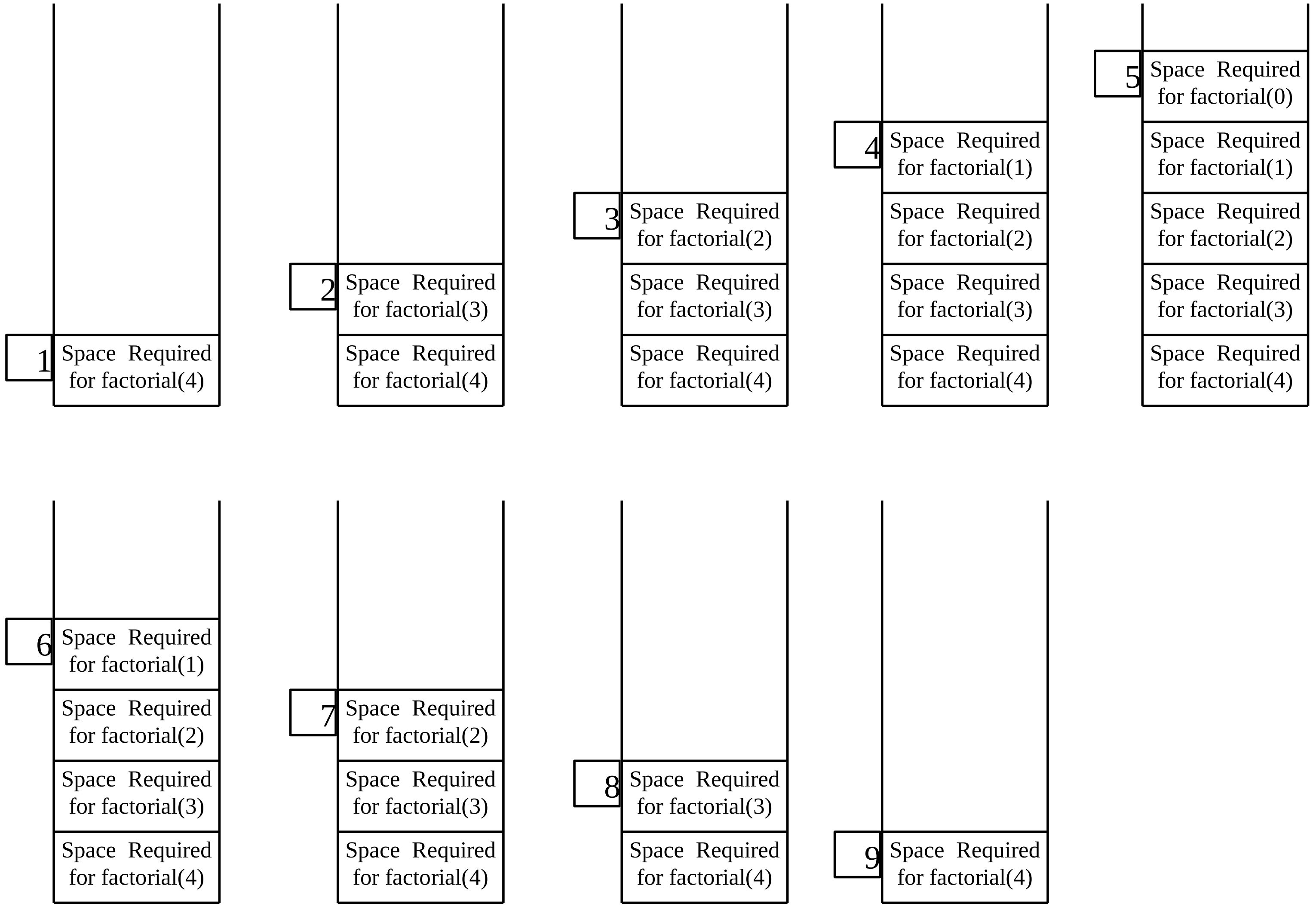


Trace Recursive factorial





factorial(4) Stack Trace



1
Space Required for factorial(4)
2
Space Required for factorial(3)
Space Required for factorial(4)
3
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
4
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
5
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
6
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
7
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
8
Space Required for factorial(3)
Space Required for factorial(4)
9
Space Required for factorial(4)

Tail Recursion

- A recursive method is said to be **tail recursive** if there are no pending operations to be performed on return from a recursive call.

Demo Program:

Non-Tail-Recursive: ComputeFactorial.java

Tail-Recursive: ComputeFactorialTailRecursion.java

Fibonacci Numbers

LECTURE 7

Recursive Method

A method calling itself.

- A recursive method is defined as a method which calls itself.
- It is realized by call-stacks. **Call-stack** keep track of which call frame is currently active. Call-stack's push and pop operations works as the branching out new method call or returning from a branch-out method call.
- Stop condition will stop a method to continue to branch out and return certain value back to the calling frame.

Fibonacci Numbers

Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…

indices: 0 1 2 3 4 5 6 7 8 9 10 11

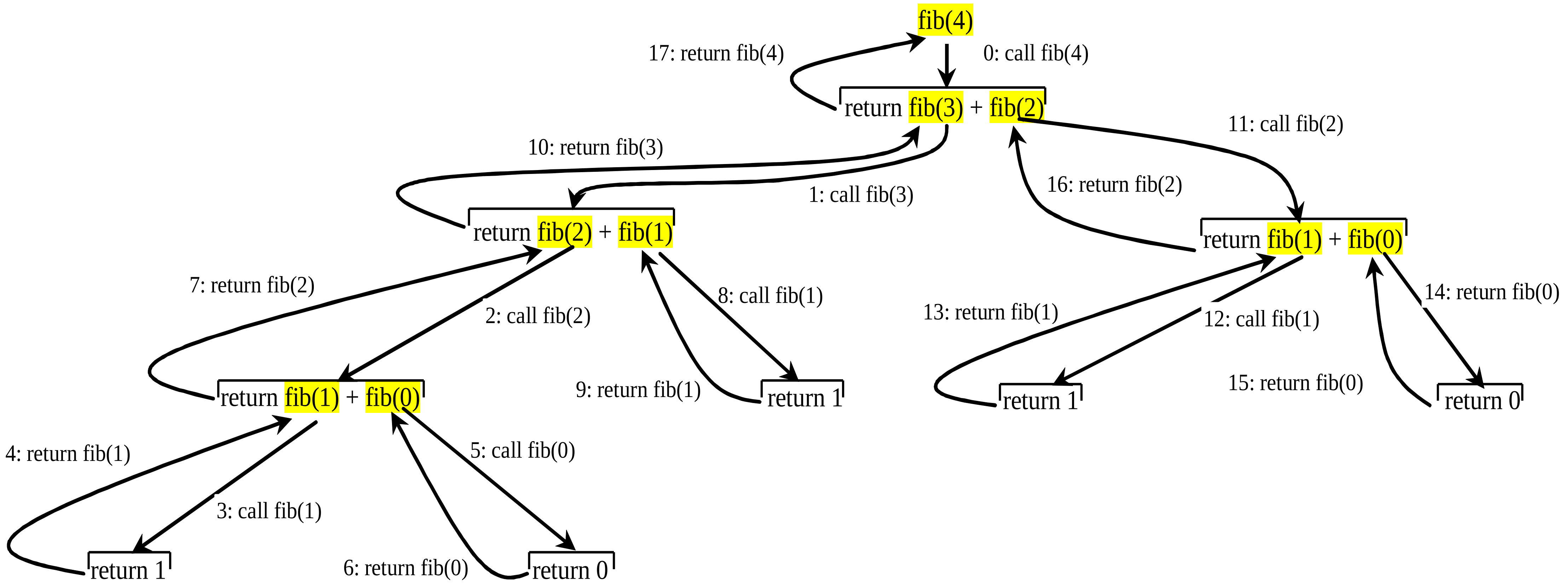
fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2

fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1) = (1 + 0) +fib(1) = 1 + fib(1) = 1 + 1 = 2

Fibonnaci Numbers, cont.



Fibonacci Method is a 2nd Order Linear Recurrence Equation

http://mathworld.wolfram.com/LinearRecurrenceEquation.html

- For recursive method which branch out two method call, it is called 2nd order recursive method.
- 2nd order recursive method may need 2 stop conditions.
- The time complexity grows in order of $O(2^n)$

Demonstration Program

COMPUTEFIBONACCI.JAVA