



LECTURE 13

Recursion

ITCS123 Object Oriented Programming

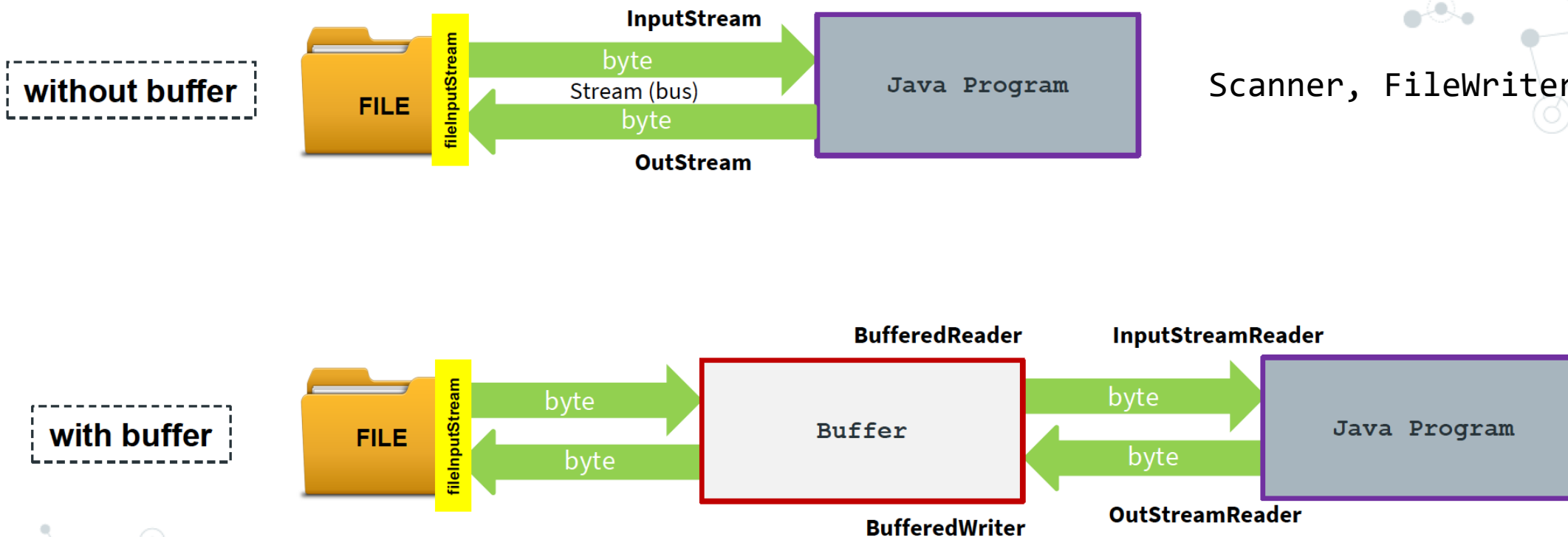
Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

(Some materials in the lecture are done by Aj. Suppawong Tuarob)

Ref: Java Concepts Early Objects by Cay Horstmann

Review: File Management

Scanner, FileWriter Class



Efficient way to read/write files. A data buffer is temporary memory that can be faster access than disk.

Writing CSV File

```
public static final String[] header = {"ID", "Name", "E-mail"};
public static final String[][] students = {
    {"6488125", "David Beckham", "dback@school.edu"},
    {"6488126", "Christina Aguilera", "caguilera@school.edu"},
    {"6488127", "Lady Gaga", "lgaga@school.edu"}
};

public static void writeCSV(String outCsvFilename, String[] header, String[][] input)
{
    CSVPrinter printer = null;
    try {
        //create a CSV printer handler
        printer = new CSVPrinter(new FileWriter(outCsvFilename), CSVFormat.DEFAULT);

        //print headers
        printer.printRecord(Arrays.asList(header));

        //print data each row
        for(String[] row: input)
        {
            printer.printRecord(Arrays.asList(row));
        }

    } catch (IOException e) {
        e.printStackTrace();
    } finally
    {
        if(printer != null) try { printer.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

```
writeCSV("test-students.csv", header, students);
```



test-students.csv

```
ID,Name,E-mail
6488125,David Beckham,dback@school.edu
6488126,Christina Aguilera,caguilera@school.edu
6488127,Lady Gaga,lgaga@school.edu
```

Reading CSV (Do not care about headers)

```
public static void readCSVSimple(String csvFilename)
{
    CSVParser csvParser = null;
    try {
        //create a parser
        csvParser = new CSVParser(new FileReader(csvFilename), CSVFormat.DEFAULT);

        //parse each row using column IDs as indexes
        for (CSVRecord record : csvParser) {
            for(int colID = 0; colID < record.size(); colID++)
            {
                System.out.print(record.get(colID)+"|");
            }
            System.out.println();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {
            if(csvParser != null) csvParser.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
readCSVSimple("small-no-headers.csv");
```

small-no-headers.csv

```
1995,Java,James Gosling
"1995","Java","James Gosling"
"March 22, 2022",Java, Quotes in "Cell"
```



Editor console

```
1995|Java|James Gosling|
1995|Java|James Gosling|
March 22, 2022|Java| Quotes in "Cell"|
```

Reading CSV with Headers

```
public static void readCSVwithHeaders(String csvFilename)
{
    CSVParser csvRecordsWithHeader = null;
    try {
        //create a parser and auto detect headers
        csvRecordsWithHeader = CSVFormat.DEFAULT.withFirstRecordAsHeader().parse(new FileReader(csvFilename));
        //Though deprecated, still usable

        List<String> headers = csvRecordsWithHeader.getHeaderNames();
        System.out.println("Detected Headers: "+headers);

        //parse each row using String headers as indexes
        for (CSVRecord record : csvRecordsWithHeader) {
            for(int colID = 0; colID < record.size(); colID++)
            {
                System.out.print(headers.get(colID)+":"+record.get(headers.get(colID))+ "|");
            }
            System.out.println();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {
            if(csvRecordsWithHeader != null) csvRecordsWithHeader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
readCSVwithHeaders("small-with-headers.csv");
```

small-with-headers.csv

```
year, language, name
1995,Java,James Gosling
"1995","Java","James Gosling"
"March 22, 2022",Python, Quotes in "Cell"
```



Editor console

```
Detected Headers: [year, language, name]
year:1995| language:Java| name:James Gosling|
year:1995| language:Java| name:James Gosling|
year:March 22, 2022| language:Python| name: Quotes in "Cell"|
```

Working with JSON Format

- Two things are required:
 - **Serialization**: encode Java Object to its JSON representation
 - **Deserialization**: decode String back to an equivalent Java Object

```
public class Course {  
    String course_code:  
    String course_name:  
    int credit;  
  
    Public Course (String c, String n, int credit) {  
        ...  
    }  
}
```

Serialization

```
{  
    "course_code": "ITCS209",  
    "course_name": "OOP",  
    "credit": 3  
}
```

Deserialization

- How to do that in Java?
 - Option 1: Write your own code to parse JSON text file **// NOT recommended**
 - Option 2: Using external Java library **// YES YES YES**

Simple Serialization GSON



```
public class Course {  
    private String name;  
    private int credit;  
    private List<String> instructors;  
  
    public Course(String name, int credit, List<String> instructors) {  
        this.name = name;  
        this.credit = credit;  
        this.instructors = instructors;  
    }  
    ...  
}  
  
// -- main method --  
Course course = new Course("OOP", 3, Arrays.asList("Siripen", "Petch", "Suppawong"));  
String serializedCourse = new Gson().toJson(course);  
System.out.println(serializedCourse);
```

OUTPUT

```
{"name":"OOP","credit":3,"instructors":["Siripen","Petch","Suppawong"]}
```

Simple Deserialization GSON

```
public class Course {
    private String name;
    private int credit;
    private List<String> instructors;

    public Course(String name, int credit, List<String> instructors) {
        this.name = name;
        this.credit = credit;
        this.instructors = instructors;
    }

    public String toString() {
        return "name=" + name + "::credit=" + credit + "::instructors=" + instructors.toString();
    }
}

// -- main method --
String courseJson = "{\\"name\\":\\"OOP\\",\\"credit\\":3," +
    "\\"instructors\\":[\\"Siripen\\",\\"Petch\\",\\"Suppawong\\"]}";

Course oopCourse = new Gson().fromJson(courseJson, Course.class);
System.out.println(oopCourse);
```

OUTPUT

name=OOP::credit=3::instructors=[Siripen, Petch, Suppawong]

Review: RegEx

My name is Siripen_Pongpaichet.
My student ID is 6288999
My phone number is 02-441-0909
My line contact is @inging99
My email is siripen.pon@mahidol.ac.th

Example Patterns

Information	Observation	Actual RegEx Patter
Name	{one or more letter}_{one or more letter}	[a-zA-Z]+_[a-zA-Z]+
Student ID	{7digits number only}	[0-9]{7}
Phone Number	{2digits}-{3digits}-{4digits}	[0-9]{2}-[0-9]{3}-[0-9]{4}
Line	@{text/number}	@[\w]+
Email	{one or more letter}.{3letter}@{text}.{th com}	[\w]+.[a-zA-Z]{3}@[\w.]+.[th com]

RegEx in Java

1. Import library (`import java.util.regex.*;`)
2. Define **RegEx** pattern in String format
3. Create and compile **Pattern** from the RegEx String in step 2
4. Create **Matcher** from the Pattern in step 3
5. *Scan the input sequence and find the subset of text that matches the pattern*

RegEx Code in Java

1

```
import java.util.regex.*;
```

```
public class Part1_BasicRegEx {
```

```
    public static void main(String[] args) {
```

```
        String text = "abcdefghabcd";
```

2

```
        String regex = "abc";
```

```
        Pattern p = Pattern.compile(regex);
```

4

```
        Matcher m = p.matcher(text);
```

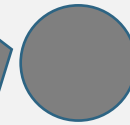
```
        System.out.println("matches the entire string: " + m.matches());
```

```
        System.out.println("matches at the beginning of string: " + m.lookingAt());
```

```
        System.out.println("matches any part of the text string: " + m.find());
```

```
    }
```

```
}
```



Text Source

3

Notice that neither Pattern nor Matcher has a public constructor; you create them using methods in the Pattern class.

5

false

true

true

Today Learning Outcomes

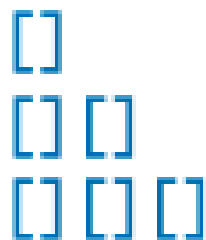
- After finished this class, students are able to
 - *Demonstrate* how to “think recursively”
 - *Explain* the relationship between recursion and iteration
 - *Implement* a program to solve the problem using recursive and recursive helper methods
- Topics
 1. Think Recursively
 2. Implement Recursive Methods
 3. Recursive Helper Methods
 4. Recursion vs Iteration
 5. More Examples



“Think Recursively”

Think like a boss (but a good boss who still do some small work)
and let employees do the rest of the work for you

Let's look at a simple example – Triangle



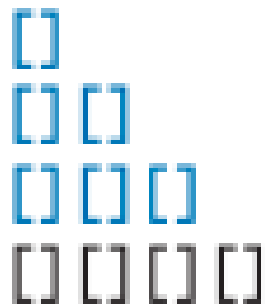
- Suppose we would like to compute area of a triangle of width n , assuming that each `[]` square has area 1.
- The area of this n^{th} triangle number (where $n=3$) shown above is 6

```
public class Triangle
{
    private int width;

    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    public int getArea()
    {
        . . .
    }
}
```

How to compute area of this Triangle



- If the width of the triangle is 1 ($n = 1$), then the triangle has only a single square, and its area is 1

```
if (width == 1) { return 1; }
```

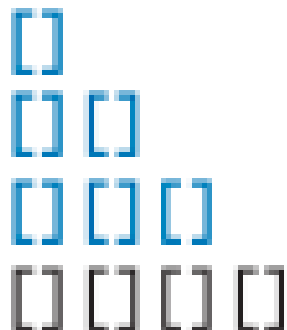
- Suppose we knew the area of the smaller, colored triangle. We can easily compute the area of the larger triangle as

`smallerArea + width`

- How can we get the smaller area? Just make a smaller triangle and **ask** it!

```
Triangle smallerTriangle = new Triangle(width - 1);
```

```
int smallerArea = smallerTriangle.getArea();
```



```
public int getArea() {  
    if (width == 1) { return 1; }  
    else {  
        Triangle smallerTriangle = new Triangle(width - 1);  
        int smallerArea = smallerTriangle.getArea();  
        return smallerArea + width;  
    }  
}
```

What actually
happends when we
compute the area of
a triangle of width 4

- The **getArea** method makes a smaller triangle of width 3.
- It calls **getArea** on that triangle.
 - That method makes a smaller triangle of width 2.
 - It calls **getArea** on that triangle.
 - That method makes a smaller triangle of width 1.
 - It calls **getArea** on that triangle.
 - That method **returns** 1.
 - The method **returns smallerArea + width** = 1 + 2 = 3.
 - The method **returns smallerArea + width** = 3 + 3 = 6.
- The method **returns smallerArea + width** = 6 + 4 = 10.

Recursive Solution

- We solve the finding area problem for a triangle of a given width, by trying to solve the same problem for a lesser width. – This is called *recursive* solution
- There are two key requirements to make this recursion successful:
 - Every recursive call must simplify the problem in some way
 - There must be special cases to handle the simplest computations directly
- In the triangle example,
 - Every recursive call try to get area of the smaller width triangle
 - Eventually, the width must reach 1 and this is a special case of area 1

Let's try more examples

How many people
standing in line now?

It helps, if you pretend to be a bit lazy, and asking others to do most of the work for you!

“Someone else” will help you solve the problem for **simpler inputs**. So you just have to figure out how to simplify the inputs for them, and how to combine their results to solve your whole problem.



Try again and again!

- Print text in reverse
 - Given text: RECURSION -> Print out: NOISRUCER
- Count number of 7 in a given number
 - Given number: 19775278 -> Return: 3
- Check whether a given word is palindrome
 - Palindrome is a string that is equal to itself when you reverse all characters
 - E.g., wow, noon, solos, stats, radar, madam, abaaccaaba
 - Return true if the given word is palindrome, and false otherwise.

Thinking Recursively - Palindrome

- **Problem Statement: Test whether a given word is a palindrome**
- **Step 1:** Consider various ways to simplify inputs
 - The given input is a string. There are many possibilities to simplify it such as
 - Remove the first character
 - Remove the last character
 - Remove both the first and the last characters
 - Remove a characters from the middle
 - Cut the string into two halves
 - Which one work best for this problem???

Thinking Recursively – Palindrome (cont.)

- **Step 2:** Combine solutions with simpler inputs into a solution of the original problem
 - Don't worry about **how** to get the solution from the simpler inputs, just have faith that the solutions from the simpler inputs are available.
 - Think about for the input that you are **how to turn the solution for the simpler inputs into a solution** holding now.
 - For example, add a small quantity to the solution, or if you cut the input into two halves, you may have to add them up
 - Suppose we simplify inputs in step 1 by removing the first and the last letter. The word is palindrome if and only if **the first and the last letter match AND the shorter word obtained by removing the first and the last letter is a palindrome.**

Thinking Recursively – Palindrome (cont.)

- **Step 3:** Find the solutions to the simplest inputs
 - A recursive computation keeps simplifying its inputs. At the end, **it arrives very simple inputs and recursion comes to a stop!**
 - This simplest inputs can possibly in many forms such as empty string, number 0, shapes with no area, null objects, etc.
 - According to what we discovered in Step 2, we can find the simplest inputs:
 - Strings with two characters - This can be handle as usual (remove first and last)
 - String with a single characters - Step 2 cannot apply (not enough string to remove)
 - The empty string - Step 2 cannot apply, no more string to compare
 - The later two cases (e.g., “a” and “”) can be consider as a palindrome word. **Thus, we can conclude that all string of length 0 or 1 are palindromes.**

Thinking Recursively – Palindrome (cont.)

- **Step 4:** Implement the solution by combining the simple cases and the reduction step.
 - Make a separate cases for the simplest inputs in Step 3.
 - If the input is not one of the simplest cases, then implement the logic we setup in Step 2.

```
public static boolean isPalindrome(String word) {  
    int length = word.length();  
  
    // Separate case for shortest strings.  
    if (length <= 1) { return true; }  
    else {  
        // Get first and last characters  
        char first = word.charAt(0);  
        char last = word.charAt(length - 1);  
        if (first == last) {  
            // Remove both first and last character.  
            String shorter = word.substring(1, length - 1);  
            return isPalindrome(shorter);  
        } else {  
            return false;  
        }  
    }  
}
```



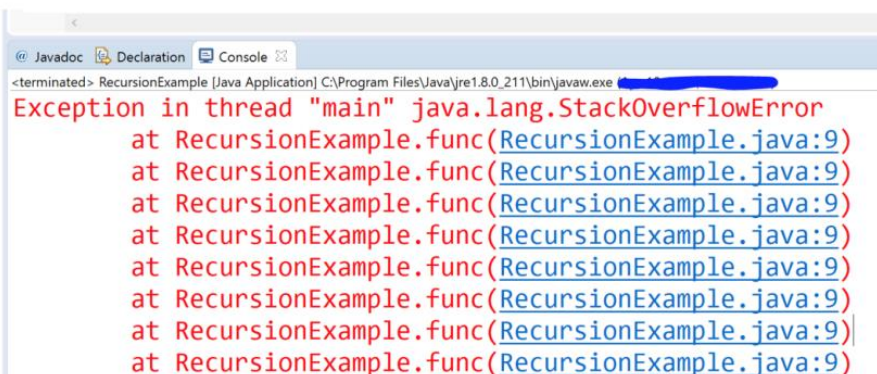
Implement Recursive Methods

To understand recursion – Must understand stack

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

```
static void func() {  
    func();  
}
```

So.. What is "stack"?



Exception in thread "main" java.lang.StackOverflowError
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)
at RecursionExample.func(RecursionExample.java:9)

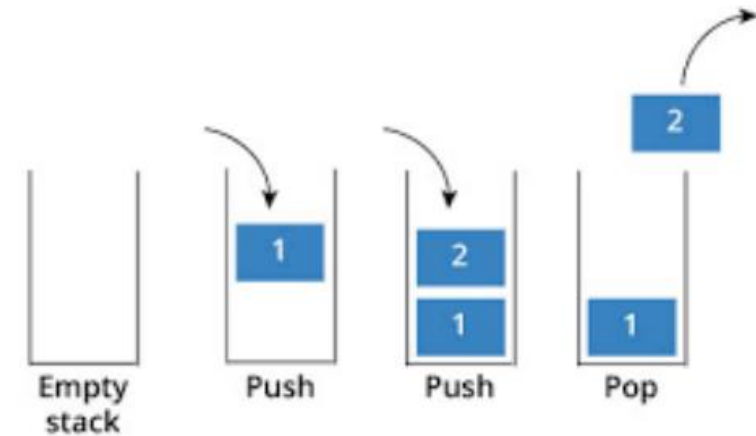
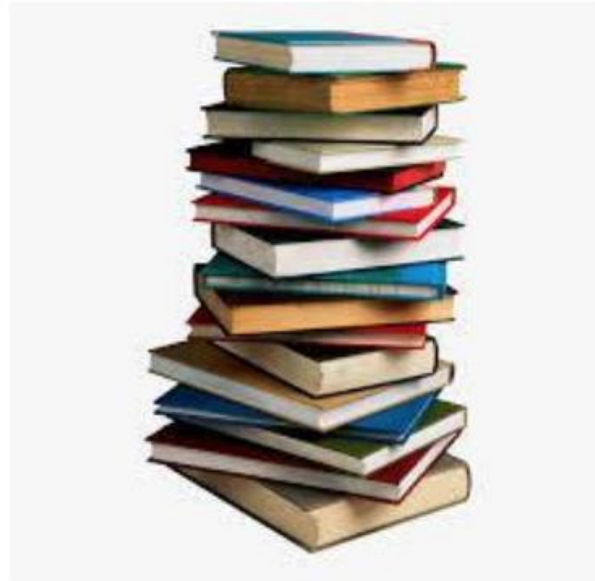


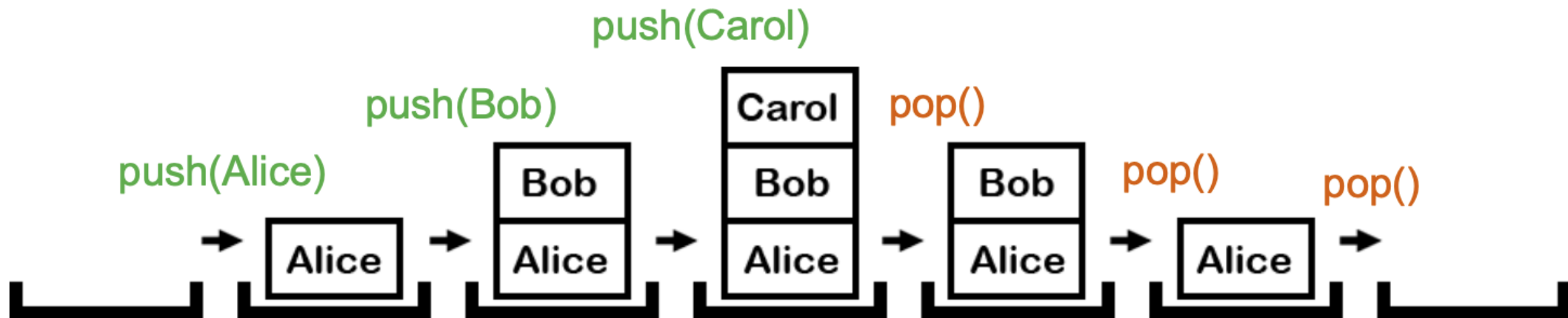
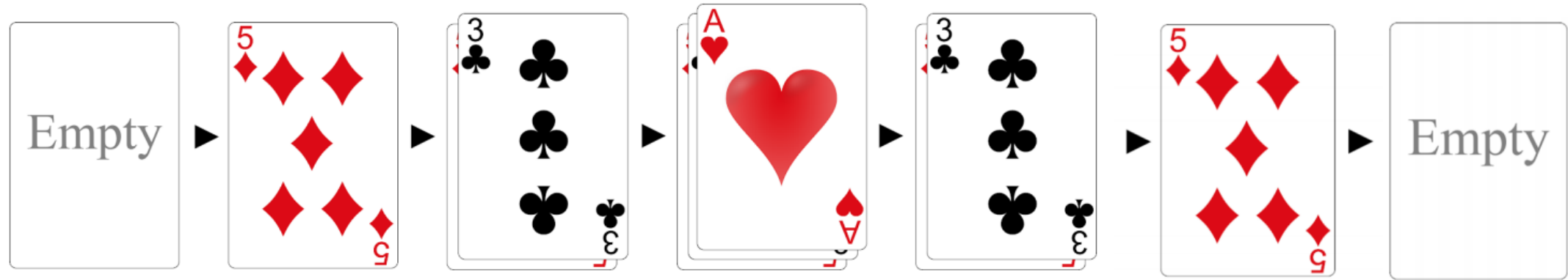
Stack Overflows

results from too much data
being pushed onto the stack.
The memory/capacity of the
stack is exceeded.



A **stack** is a data structure that holds a sequence of data and only lets you interact with the **topmost** item.





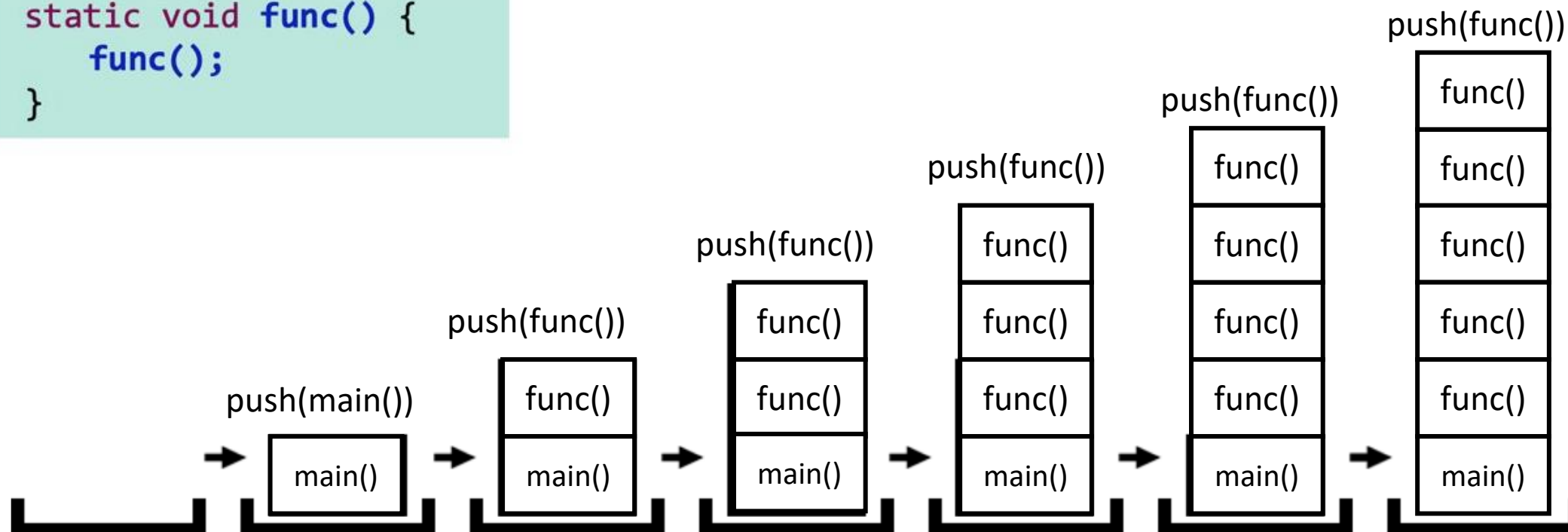
First-In, Last-Out (FILO)

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

```
static void func() {  
    func();  
}
```

None stop putting into the stack!

--> eventually stack overflows

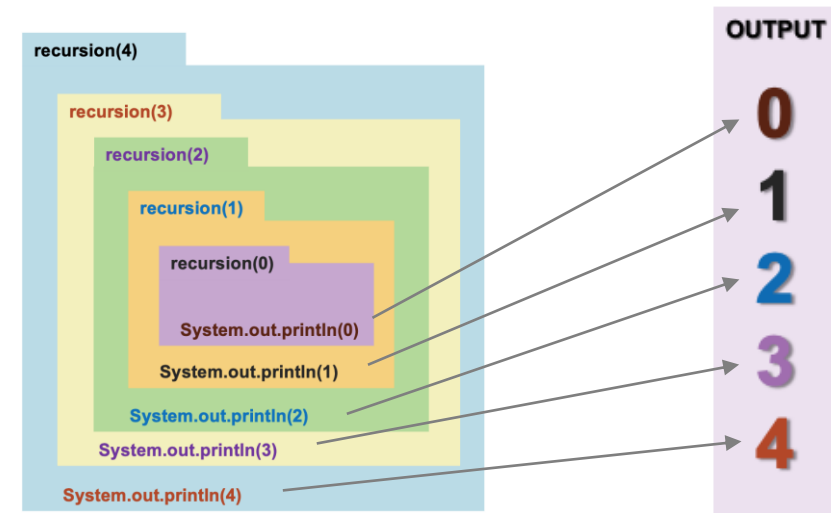
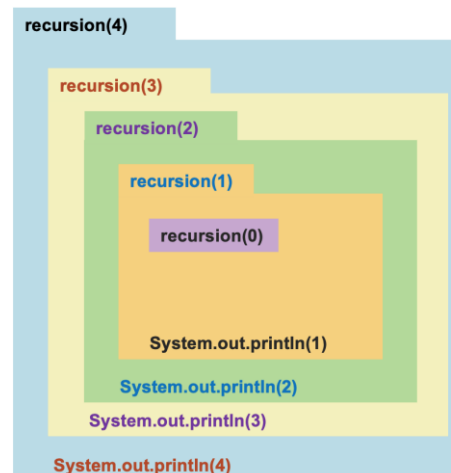
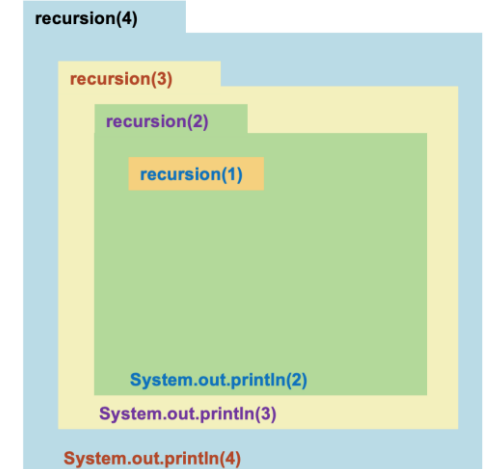
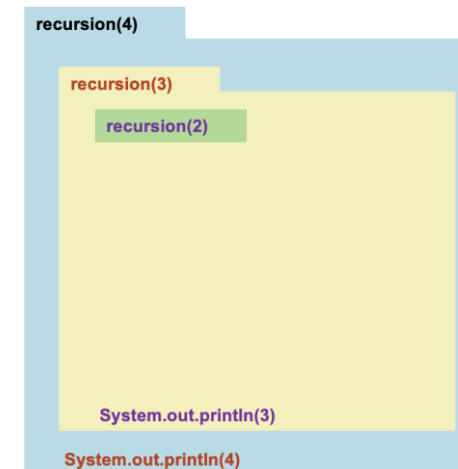
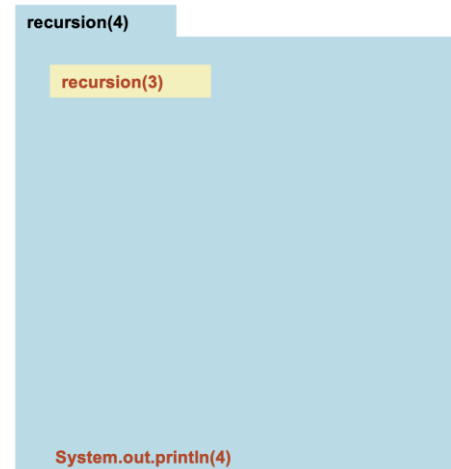


How about recursion method?

```

public static void recursion(int x){
    if(x == 0){
        System.out.println(x);
    } else{
        recursion(x-1);
        System.out.println(x);
    }
}
    
```

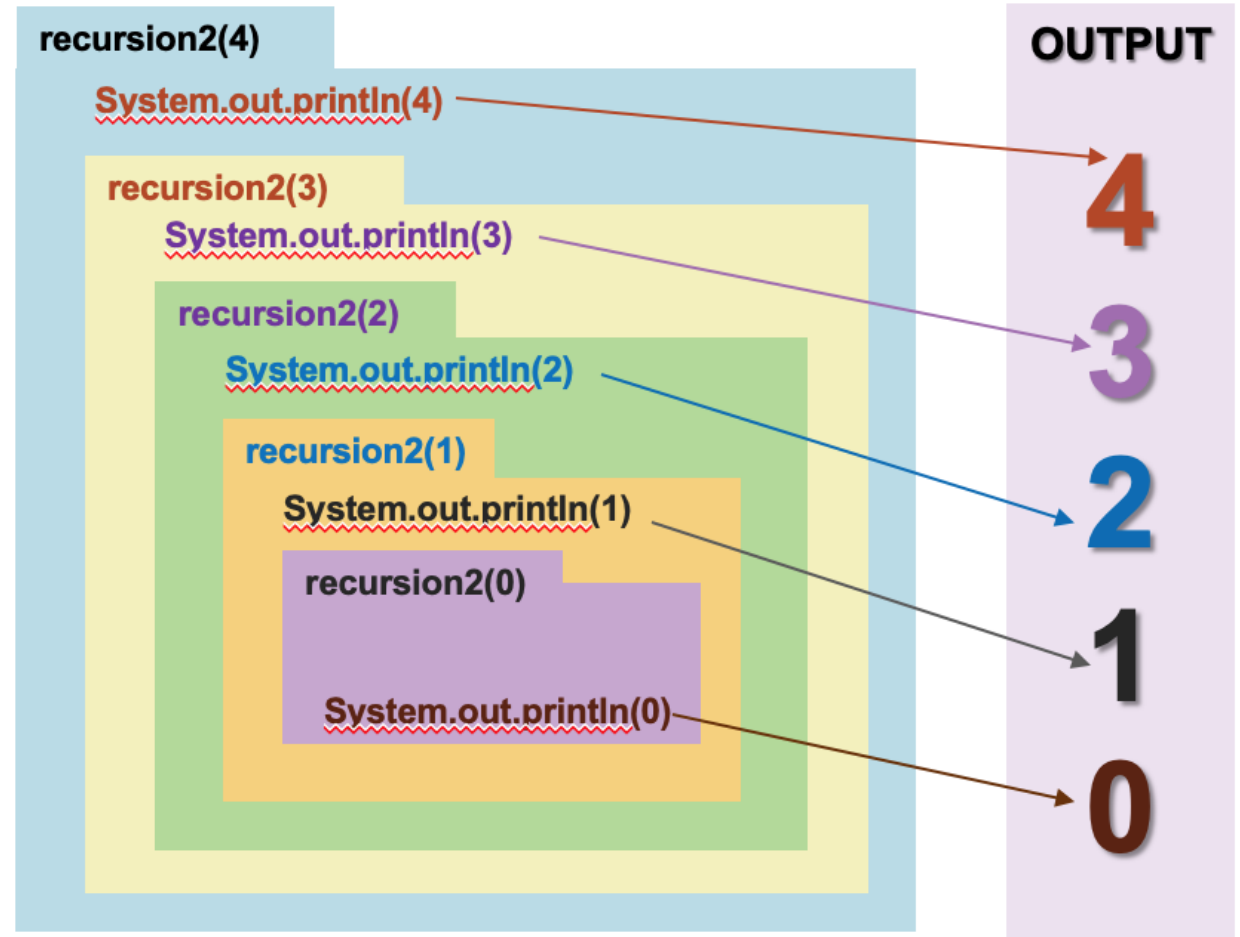
What is the output of
recursion(4)?



Another Example

```
static void recursion2(int x){  
    if(x == 0){  
        System.out.println(x);  
    } else{  
        System.out.println(x);  
        recursion2(x-1);  
    }  
}
```

recursion2(4);





Classic Example: Factorial (!)

Factorial has a recursive nature

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Factorial Recursive Method

Number! = Number × (Number - 1)!

```
static int factorial(int n) {
    return n * factorial(n-1);
}

public static void main(String[] args) {
    System.out.println(factorial(5));
}
```

Q: Is this factorial(n) method correct?

Press 1, if you think it is perfect!

Press 2, if you think it will have a run-time error

Press 3, if you think it will have a logical error



Runtime Error: StackOverflowError

```
3
4 public static void main(String[] args) {
5     System.out.println(factorial(5));
6 }
7
8 static int factorial(int n) {
9     return n * factorial(n-1);
10 }
11 }
```

[illegible]

Revisit Factorial Recursive Method

$$\text{Number!} = \text{Number} \times (\text{Number} - 1)!$$

Expected

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

But we actually did

$$5! = 5 \times 4 \times 3 \times 2 \times 1 \times 0 \times -1 \times -2 \times \dots = 120$$

A stack overflow error is when a recursive function gets out of control and **does not stop recursing**. As a result, the stack is full and cannot accept any more method call.

So when should it stop?

Revisit Factorial Recursive Method

$$\text{Number!} = \text{Number} \times (\text{Number} - 1)!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

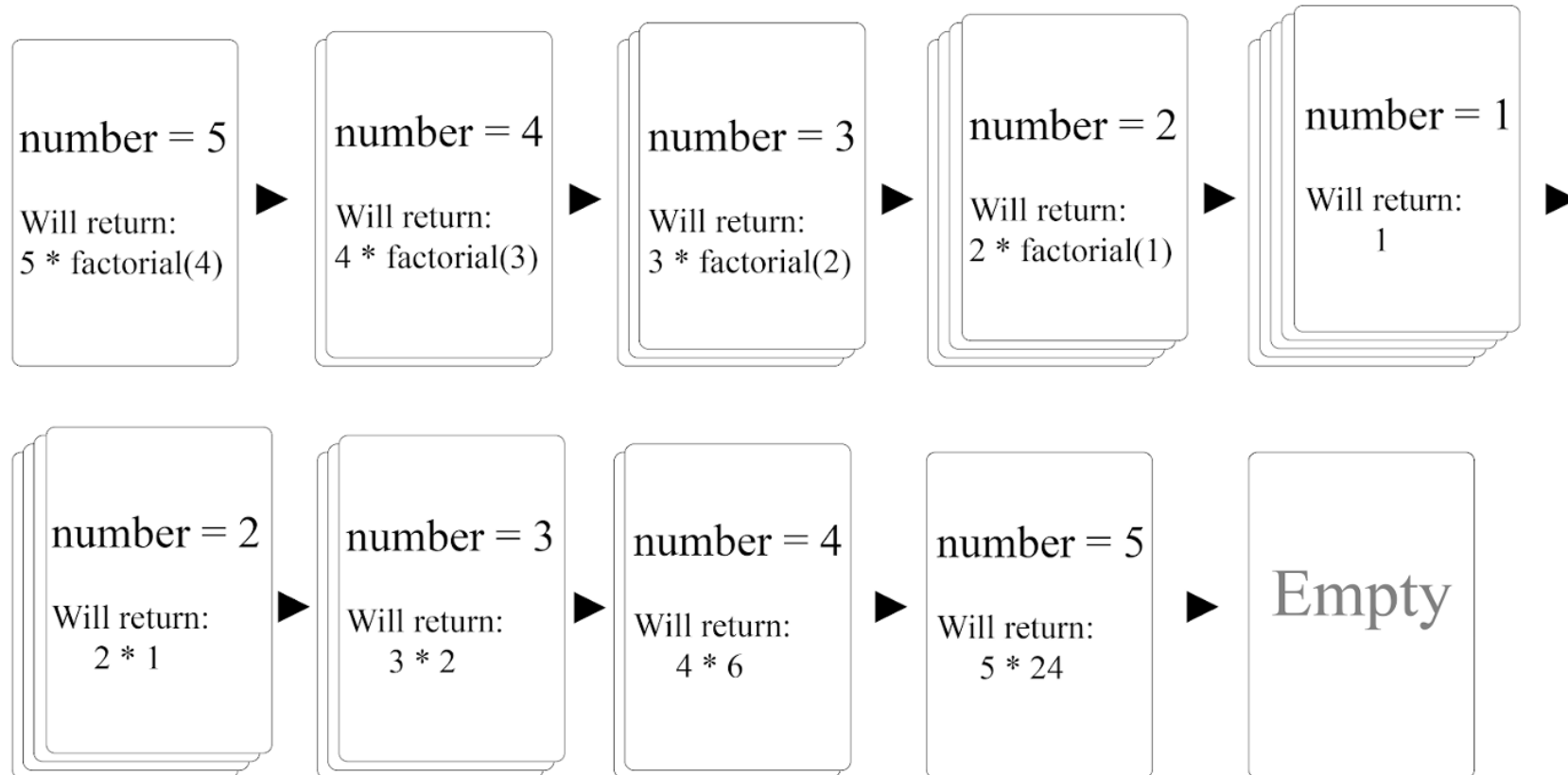
$$1! = 1$$

```
static int factorial(int n) {  
    if(n == 1) // BASE CASE  
        return 1;  
    else // RECURSIVE CASE  
        return n * factorial(n-1);  
}
```

Your recursive method must always have
at least one base case and one recursive case.

```
static int factorial(int n) {  
    if(n == 1)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

“Where does the $5 \times 4 \times 3 \times 2 \times 1$ happen?”



Hard-coded pseudo-recursive algorithm

```
public static void main(String[] args) {
    System.out.println(factorial5());
}

static int factorial5() {
    return 5 * factorial4();
}
static int factorial4() {
    return 4 * factorial3();
}
static int factorial3() {
    return 3 * factorial2();
}
static int factorial2() {
    return 2 * factorial1();
}
static int factorial1() {
    return 1;
}
```

Recursive factorial algorithm

```
static int factorial(int n) {
    if(n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

Iterative factorial algorithm

```
public static void main(String[] args) {
    System.out.println(factorialLoop(5));
}

static int factorialLoop(int n) {
    int result = 1;
    for(int i = 1; i <= 5; i++) {
        result = result * i;
    }
    return result;
}
```



Recursive Helper Methods

Recursive Helper Methods

- To find recursive solution, sometimes it is easier to slightly change the original problem. Then let the original problem call a **recursive helper methods**.
- For palindrome problem, it is inefficient to construct a new string in every step. We can change a little bit to check whether a substring is a palindrome.

```
/**  
    Tests whether a substring is a palindrome.  
    @param text a string that is being checked  
    @param start the index of the first character of the substring  
    @param end the index of the last character of the substring  
    @return true if the substring is a palindrome  
*/  
public static boolean isPalindrome(String text, int start, int end)
```

```
public static boolean isPalindrome2(String word) {  
    return isPalindrome2(word, 0, word.length() - 1);  
}  
  
public static boolean isPalindrome2(String word, int start, int end){  
    // Separate case for substring of length 0 and 1  
    if (start >= end) { return true; }  
    else {  
        // Get first and last characters  
        char first = word.charAt(start);  
        char last = word.charAt(end);  
        if (first == last) {  
            // Test substring that doesn't contain the matching letters.  
            return isPalindrome2(word, start + 1, end - 1);  
        } else{  
            return false;  
        }  
    }  
}
```

Note that the original `isPalindrome2(String)` method is NOT a recursive method. But this method calls the helper method `isPalindrome2(String, int, int)`. This helper method is a recursive method.



Recursion vs Iteration

Loop vs Recursion in Real Life: Lunch Invitation

Loop in real life



See you in 10 minutes



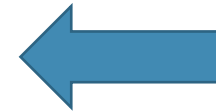
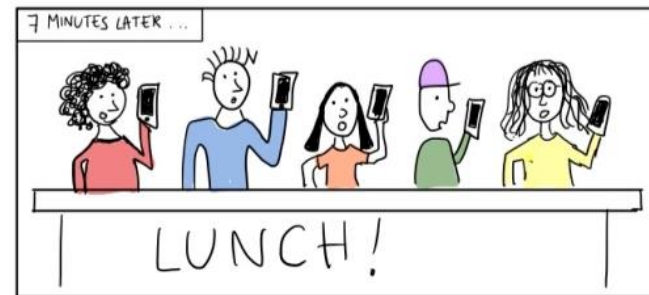
See you in 9 minutes



See you in 8 minutes



See you in 7 minutes



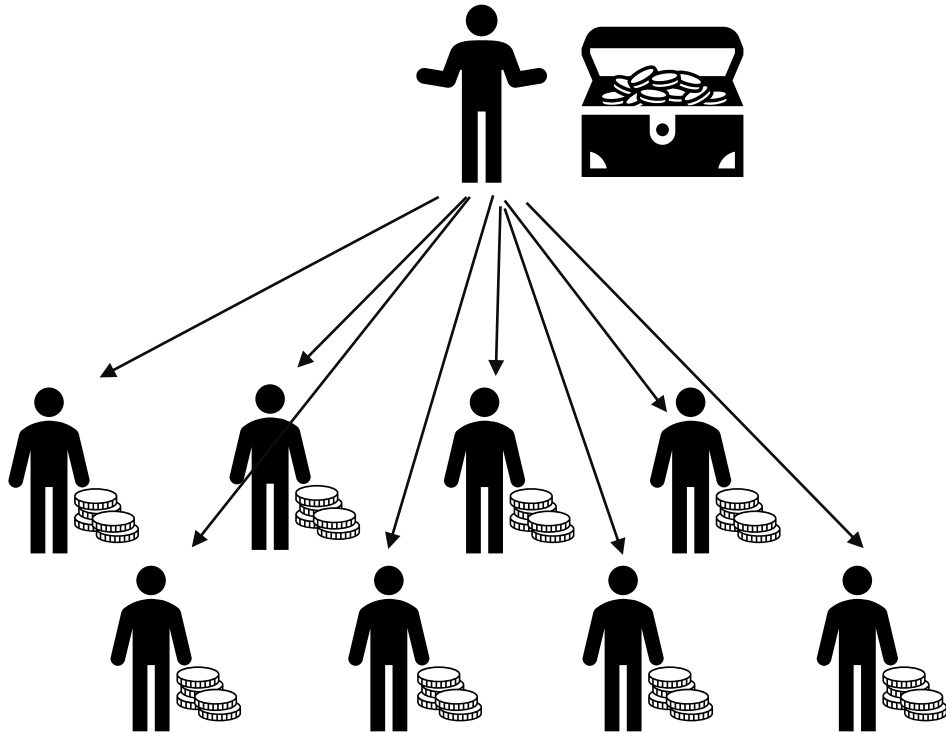
Recursion in real life



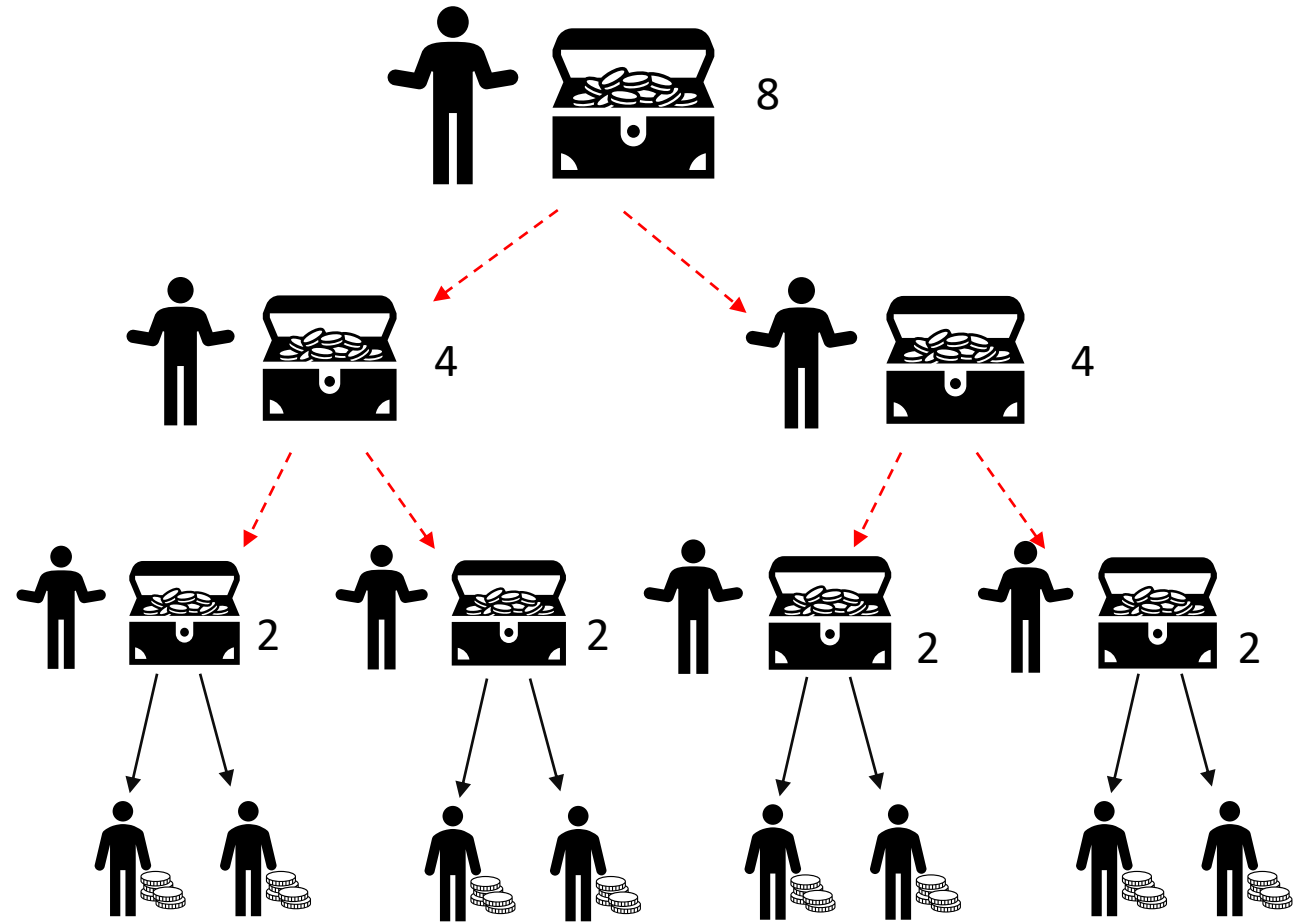
When your friend wants to invite others, then your conversation will be on hold.

Until the last person decides to go and ends the call (no more asking), everyone returns to their previous conversation one by one in sequence

Loop vs Recursion in Real Life: Fund Raising



Only **ONE** person asks **8 people** to donate money one by one



Each person ask **two more people** to donate money

Loop vs Recursion

```
public static boolean isPalindromeLoop(String word){
    int start = 0;
    int end = word.length() - 1;
    while(start < end){
        // Get first and last characters
        char first = word.charAt(start);
        char last = word.charAt(end);

        // Both match, keep continue to the next pair
        if (first == last){
            start++;
            end--;
        } else {
            return false;
        }
    }
    return true;
}
```

```
public static boolean isPalindrome(String word) {
    int length = word.length();

    // Separate case for shortest strings.
    if (length <= 1) { return true; }
    else {
        // Get first and last characters
        char first = word.charAt(0);
        char last = word.charAt(length - 1);
        if (first == last) {
            // Remove both first and last character.
            String shorter = word.substring(1, length - 1);
            return isPalindrome(shorter);
        } else{
            return false;
        }
    }
}
```



More Examples

Summation Recursive Method

```
public static int sum(int n){  
    System.out.println("start sum " + n);  
    if(n == 1){  
        System.out.println("terminate at 1");  
        return 1;  
    }  
    else{  
        int result = n + sum(n-1);  
        System.out.println("end sum " + n  
                            + " [" + result + "]" );  
        return result;  
    }  
}
```

```
public static void main(String[] args){  
    int s = sum(4);  
    System.out.println("final " + s);  
}
```

---- OUTPUT ----

main

s = sum(4)

Stack Frame

pow: Raising a Number to a Power

$$\begin{aligned}\text{E.g., } 4^4 &= 4 * 4^3 \\ &= 4 * 4 * 4^2 \\ &= 4 * 4 * 4 * 4^1\end{aligned}$$

When should it STOP ???
What value should be smaller
in each call?

```
double pow(double a, int n)
{
    if (n == 0)           // 1 base case
        return 1.0;
    else
        return a * pow(a, n-1);
}
```

Function f using Recursion

$$f(x) = \begin{cases} 1, & x = 0 \\ x * f(x - 1), & x > 0 \end{cases}$$

What is the output of f(4)?

Note that x must ≥ 0

What is the base case? What is the recursive case?

Output:
24

```
public static int f(int x){  
    if(x == 0)  
        return 1;  
    else  
        return x * f(x - 1);  
}
```

Fibonacci Series

The Fibonacci numbers, named after the Italian mathematician **Leonardo Fibonacci** (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

- If $n = 0$ then $\text{Fib}(n) = 0$
- If $n = 1$ then $\text{Fib}(n) = 1$
- If $n \geq 2$ then $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$

```
public static int fib(int n){  
    if (n == 0)  
        return 0;  
    else if (n == 1)  
        return 1;  
    else  
        return fib(n - 1) + fib(n - 2);  
}
```

// 2 base cases

Proof Rules and Bad Factorials

```
int factorial (int n)
{
    if (n == 0)
        return 0;
    else
        return n*factorial(n-1);
}
```

//0! is not 0;

always returns the wrong answer;

In the first method, the wrong answer (0) is returned for the base case; since everything depends on the base case, ultimately this method always returns 0

```
int factorial (int n)
{
    if (n == 0)
        return 1;
    else
        return factorial(n+1)/(n+1);
}
```

//n+1 not closer to 0

never returns an answer

In the second method, n+1 is farther away from the base case: this method will continue calling factorial with ever larger arguments, until the maximum int value is exceeded: a runaway (or “infinite”) recursion (actually, each recursive call can take up some space, so eventually memory is exhausted).

```
int factorial (int n)
{
    if (n == 0)
        return 1;
    else
        return n + factorial(n-1);
}
```

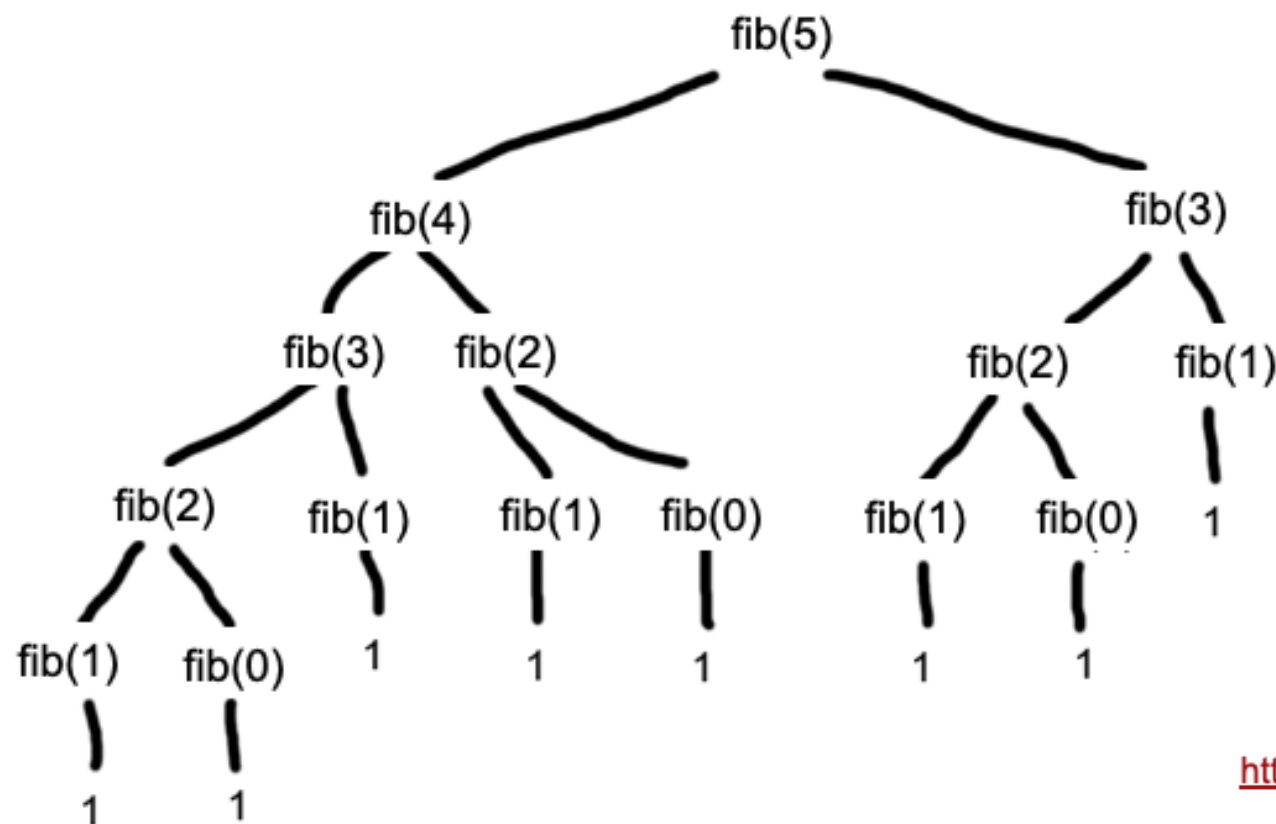
//n+(n-1)! is not n!

returns the correct answer only in the base case

In the third method, the wrong answer is returned by incorrectly combining n and the solved subproblem; this method returns one more than the sum of all the integers from 1 to n (an interesting method in its own right) not the product of these values

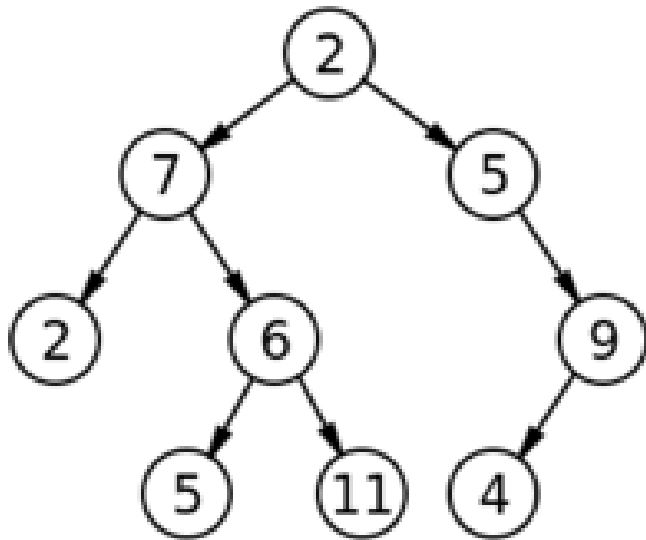
When should we use recursion?

- When the problem has a **tree-like structure**.
- When the problem requires **backtracking**.

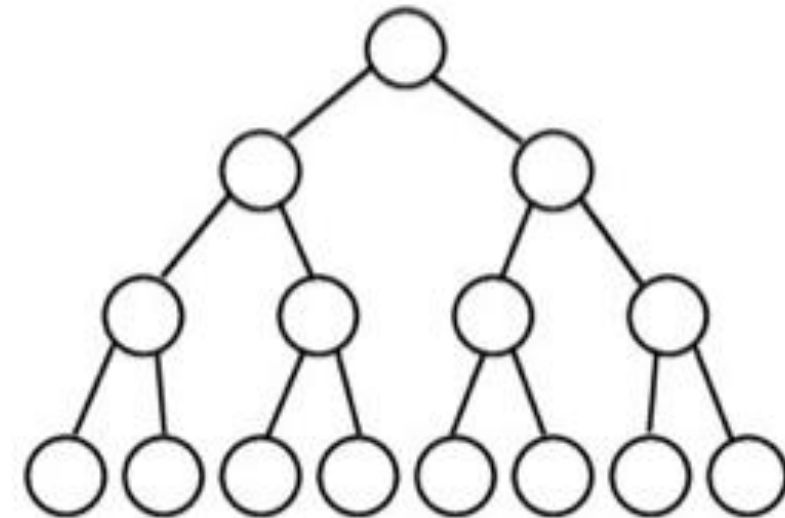


Binary Tree Data Structure

Binary tree: is a type of data structure that collect data in tree structure. Each node can have at most two children which are left child and right child.



Full binary tree: is a tree in which every node other than the leaves has two children. ?



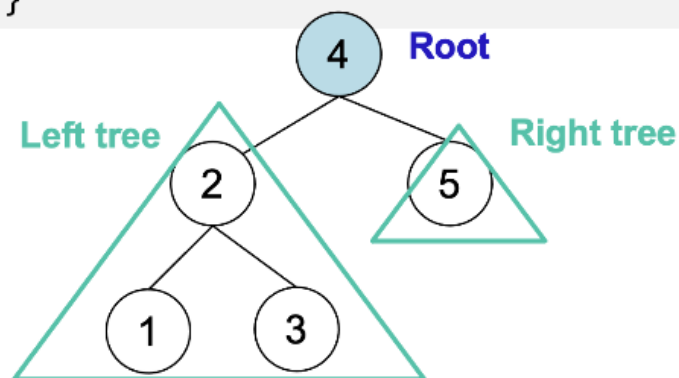
```
class Node {
    public int id;
    public Node left = null;
    public Node right = null;

    public Node(int _id, Node _left, Node _right){
        id = _id;
        left = _left;
        right = _right;
    }
}
```

```
public static void main(String[] args) {
    // construct binary tree
    Node n1 = new Node(1, null, null); // leaf node
    Node n3 = new Node(3, null, null); // leaf node
    Node n2 = new Node(2, n1, n3);     // sub-tree
    Node n5 = new Node(5, null, null); // another leaf node
    Node btree = new Node(4, n2, n5);  // whole btree

    System.out.println("Root tree ID: " + btree.id);

    //To get left and right Node, just access left and right instance fields
    System.out.println("Left tree object: " + btree.left);
    System.out.println("Right tree object: " + btree.right);
    System.out.println("Root of left tree ID: " + btree.left.id);
    System.out.println("Root of right tree ID: " + btree.right.id);
}
```



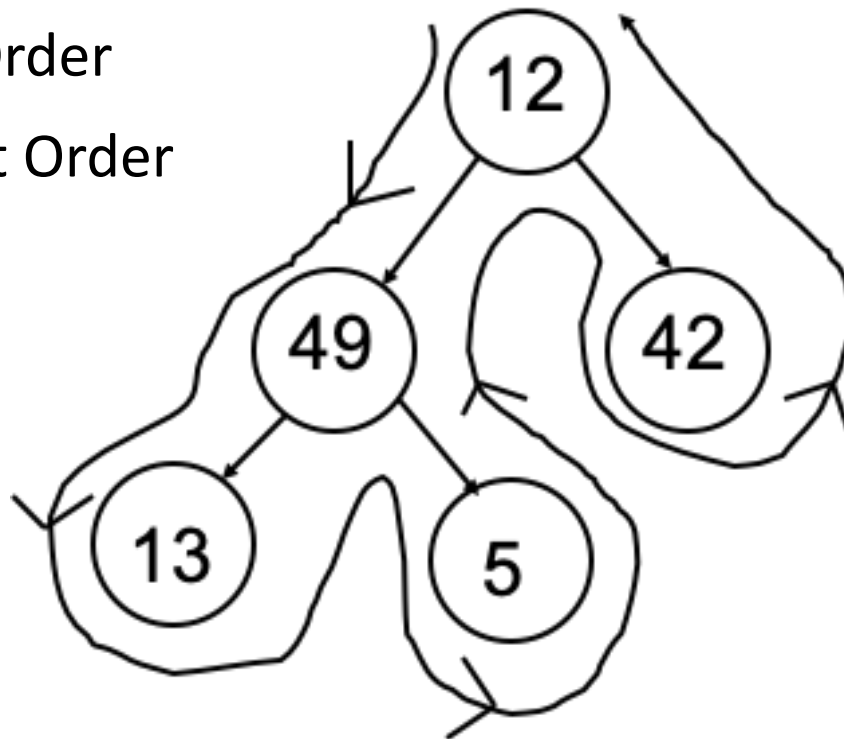
Output

```
Root tree ID: 4
Left tree object: Node@15db9742
Right tree object: Node@6d06d69c
Root of left tree ID: 2
Root of right tree ID: 5
```

Binary Tree Traversal

- You can transverse the tree recursively in one of the following orders:

- Pre Order
- In Order
- Post Order



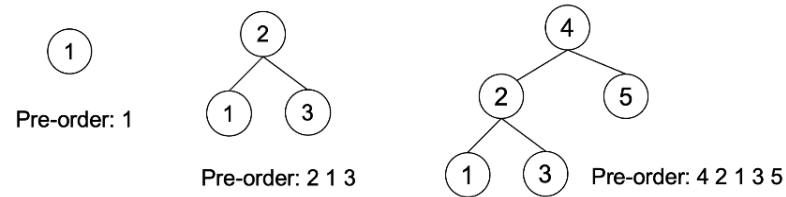
pre order: process when
pass down left side of node
4 2 1 3 5

in order: process when
pass underneath node
1 2 3 4 5

post order: process when
pass up right side of node
1 3 2 5 4

Traverse a binary tree in Pre-order

- You have to traverse through each Node in the binary tree. When traversal reaches a leaf node, it's left and right node will be NULL.
- Pre-order:** Process the root node, and pass down left side of the node then right side.

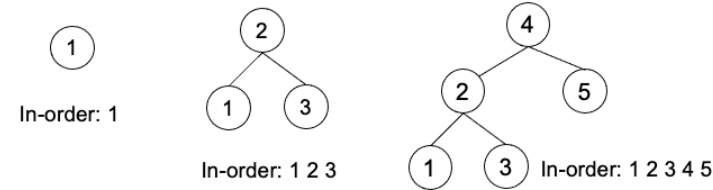


```

static void preorder(Node root) {
    if(root == null)
        return;
    else {
        System.out.print(root.id + " ");
        preorder(root.left);
        preorder(root.right);
    }
}
    
```

Traverse a binary tree in IN-order

- In-order:** process the left side of the node, then back to process the root node, and then continue to the right side of the node.

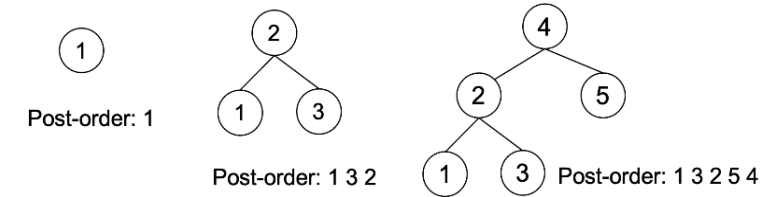


```

static void inorder(Node root) {
    if(root == null)
        return;
    else {
        inorder(root.left);
        System.out.print(root.id + " ");
        inorder(root.right);
    }
}
    
```

Traverse a binary tree in Post-order

- Post-order:** process the left side of the node, then the right side, and finally back to process the root node.



```

static void postorder(Node root) {
    if(root == null)
        return;
    else {
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.id + " ");
    }
}
    
```