

# syllabus

31 December 2021 17:20

## Introduction to Programming

- Types of languages
- Memory management

## Flow of the program

- Flowcharts
- Pseudocode

## Introduction to Java

- Introduction
- How it works
- Setup Installation
- Input and Output in Java
- Conditionals & Loops in Java
  - if-else
  - loops
  - Switch statements
- Data-types
- Coding best practices

## Functions

- Introduction
- Scoping in Java
- Shadowing
- Variable Length Arguments
- Overloading

## Arrays

- Introduction
- Memory management
- Input and Output
- ArrayList Introduction
- Searching
  - Linear Search
  - Binary Search
  - Modified Binary Search
  - Binary Search on 2D Arrays
- Sorting
  - Insertion Sort

- [Selection Sort](#)
- [Bubble Sort](#)
- [Cyclic Sort](#)

[Pattern questions](#)

[Strings](#)

- [Introduction](#)
- [How Strings work](#)
- [Comparison of methods](#)
- [Operations in Strings](#)
- [StringBuilder in java](#)

[Maths for DSA](#)

- [Introduction](#)
- [Complete Bitwise Operators](#)
- [Range of numbers](#)
- [Prime numbers](#)
- [Sieve of Eratosthenes](#)
- [Newton's Square Root Method](#)
- [Factors](#)
- [Modulo properties](#)
- [Number Theory](#)
- [HCF / LCM](#)
- [Euclidean algorithm](#)

[Recursion](#)

- [Introduction](#)
- [Flow of recursive programs - stacks](#)
- [Why recursion?](#)
- [Tree building of function calls](#)
- [Tail recursion](#)
- Sorting:
  - Merge Sort
  - Quick Sort
- Backtracking
  - Sudoku Solver
  - N-Queens
  - N-Knights
  - Maze problems
- Recursion String Problems
- Recursion Array Problems
- Recursion Pattern Problems
- Subset Questions

## Space and Time Complexity Analysis

- Introduction
- Comparisons of various cases
- Solving Linear Recurrence Relations
- Solving Divide and Conquer Recurrence Relations
- Big-O, Big-Omega, Big-Theta Notations
- Little Notations
- Get equation of any relation easily - best and easiest approach
- Complexity discussion of all the problems we do
- Space Complexity
- Memory Allocation of various languages
- NP-Completeness Introduction

## Object Oriented Programming

- Introduction
- Classes & its instances
- this keyword in Java
- Properties
  - Inheritance
  - Abstraction
  - Polymorphism
  - Encapsulation
- Overloading & Overriding
- Static & Non-Static
- Access Control
- Interfaces
- Abstract Classes
- Singleton Class
- final, finalize, finally
- Object Class
- Generics
- Exception Handling
- Collections Framework
- Lambda Expression
- Enums
- Fast IO
- File handling

## Greedy Algorithms

## Stacks & Queues

- Introduction
- Interview problems
- Push efficient

- Pop efficient
- Queue using Stack and Vice versa
- Circular Queue

- Linked List

- Introduction
- Fast and slow pointer
- Cycle Detection
- Single and Doubly LinkedList
- Reversal of LinkedList

- Dynamic Programming

- Introduction
- Recursion + Recursion DP + Iteration + Iteration Space Optimized
- Complexity Analysis
- 0/1 Knapsack
- Subset Questions
- Unbounded Knapsack
- Subsequence questions
- String DP

- Trees

- Introduction
- Binary Trees
- Binary Search Trees
- DFS
- BFS
- AVL Trees
- Segment Tree
- Fenwick Tree / Binary Indexed Tree

- Square Root Decomposition

- Heaps

- Introduction
- Theory
- Priority Queue
- Heapsort
- Two Heaps Method
- k-way merge
- Top k elements
- Interval problems

- HashMap

- Introduction
- Theory - how it works
- Comparisons of various forms
- Limitations and how to solve
- Map using LinkedList
- Map using Hash
- Count Sort
- Radix Sort
- Chaining
- Probing
- Huffman-Encoder
  
- Subarray Questions: Sliding window, Two Pointer, Kadane's Algorithm
  
- Tries
  
- Graphs
  - Introduction
  - BFS
  - DFS
  - Working with graph components
  - Minimum Spanning Trees
  - Kruskal Algorithm
  - Prims Algorithm
  - Dijkstra's shortest path algorithm
  - Topological Sort
  - Bellman ford
  - A\* pathfinding Algorithm
  
- Advanced concepts apart from interviews

Bitwise + DP  
Extended Euclidean algorithm  
Modulo Multiplicative Inverse  
Linear Diophantine Equations  
Matrix Exponentiation  
Mathematical Expectation  
Fermat's Theorem  
Wilson's Theorem  
Lucas Theorem  
Chinese Remainder Theorem  
NP-Completeness

# Roadmap

03 February 2022 10:04



✗

# DSA plan

08 March 2022 13:31

1. Kunal kush...
2. Leetcode problems

# shortcut

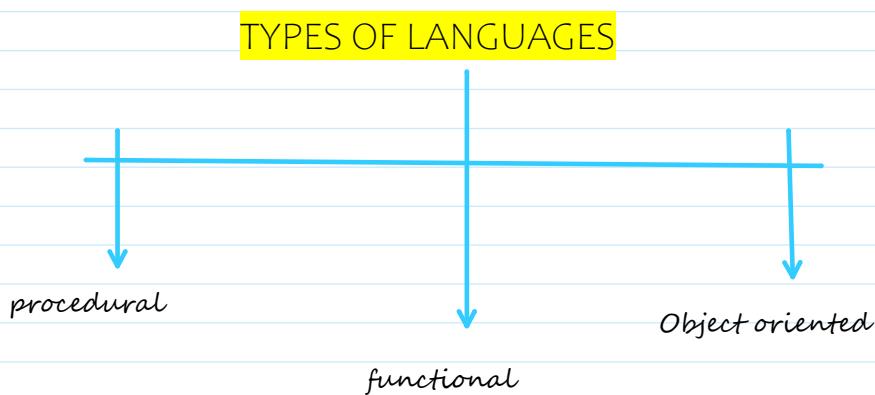
05 October 2022 16:28

- Shift+Ctrl+Alt+J --> to remove multiple carets of same name

# Types of language

31 December 2021 17:22

Programming languages -  
when we have sort of idea  
and we write in the form  
of code and then we get  
the output for that.



## Procedural :-

- ~ specifies a series of well-structured steps and procedures to compose a program.
- ~ Contains a systematic order of statements, function and command to complete task.

## Functional :-

- ~ Writing a program only in pure function i.e. never modify variable, but only create new ones as an output.
- ~ Used in situations where we have to perform lots of different operations on the same set of data, like M.L.
- ~ First class Functions ?

## Object Oriented:-

- ~ Revolves around objects .
- ~ Code + Data = object .
- ~ Developed to make it easier, debug, reuse, and maintain software .

- Python follows all three procedural, functional and object oriented whereas Java follows procedural and object oriented -----> If you're thinking that one particular language can be of one type IT'S NOT CORRECT .

# STATIC VS DYNAMIC LANGUAGES

## Static

- Perform type checking at compile time
- Errors will show at compile time
- Declare datatype before you use it
- More control

## Dynamic

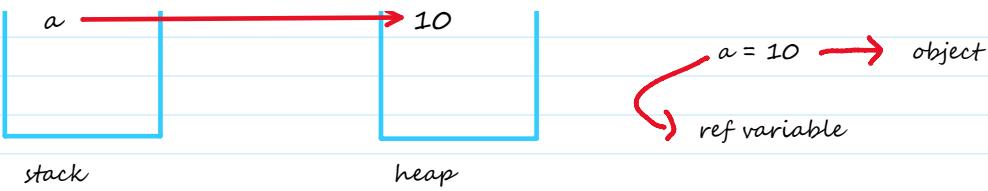
- Perform type checking at runtime .
- Errors might not show till program is run
- No need to declare datatype of variables
- Saves time in writing code but might give error at runtime

(This shows how memory is managed)

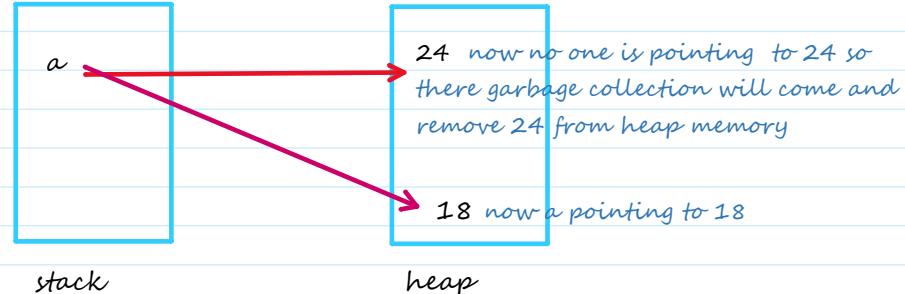
variable in stack memory pointing  
towards the object in heap memory



$a = 10 \rightarrow$  object



`a = 24;  
a=18; // now a is 18`



## FLOWCHARTS

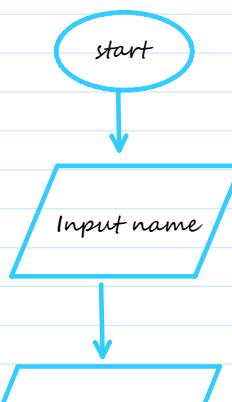
Flowcharts are basically used to visualize our particular thought process

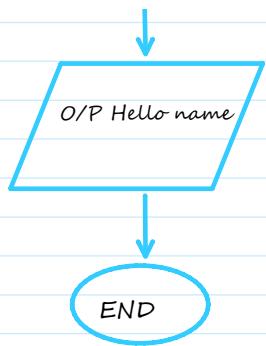


### ➤ Pseudocode

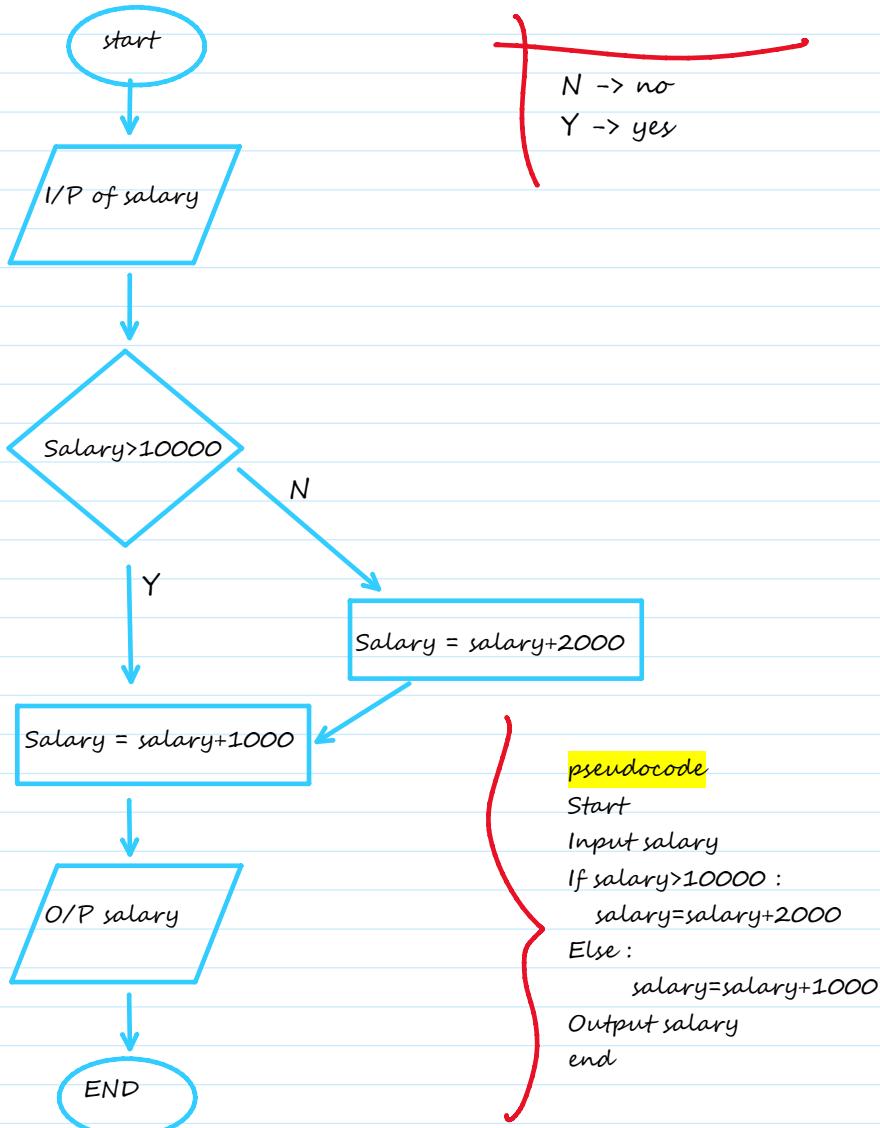
Pseudocode is an artificial and informal language that helps programmers develop algorithms.  
Pseudocode is a "text-based" detail (algorithmic) design tool.

### ➤ Example 1 ~ Take a name and output hello name.

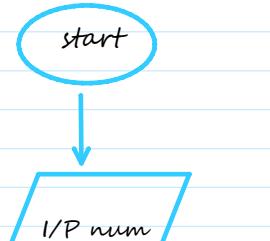




- Example 2 ~ Take input of a salary. If the salary is greater than 10,000 then add bonus as 2000, otherwise add bonus as 1000.

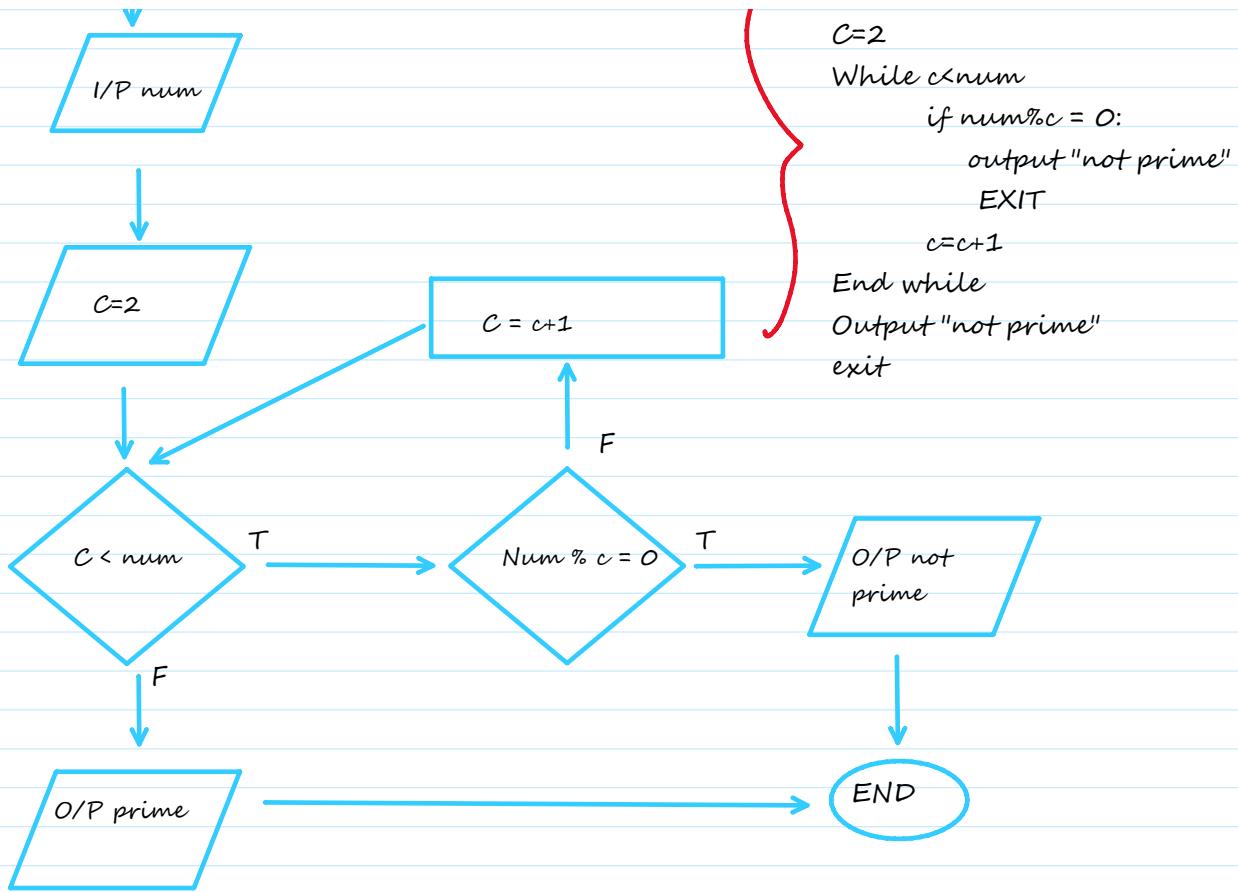


- Example 3 ~ Input a number and check whether it is prime or not



Pseudocode :-->

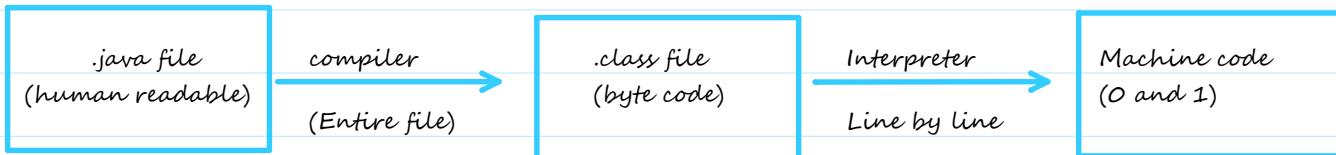
Start  
Input num  
 $C=2$   
While  $c < num$



# Introduction of Java Architecture

29 August 2021 12:21

## ➤ How java code executes



This is source code

- This code will not directly run on a system
- We need JVM to run this.
- Reason why java is platform independent

## ➤ More about platform independence

- It means that byte code can run on all operating system.
- We need to convert source code to machine so computer can understand
- Compiler helps in doing this by turning it into executable code
- This executable code is a set of instruction for the computer
- After compiling c / c++ code we get .exe file which is platform dependent.
- In java we get bytecode, JVM converts this into machine code
- Java is platform-independent but JVM is platform dependent

## ➤ JDK VS JRE VS JVM VS JIT

JDK = JRE + Development tools  
Java development kit

JRE = JVM + library classes  
Java runtime environment

Java virtual machine

JIT  
(just-in-time)

## ➤ JDK

- Provides environment to develop and run the program
- It is a package that includes:

1. development tools - to provide an environment to develop your program
2. JRE - to execute your program
3. A compiler - javac
4. Archiver - jar
5. Documentation generator - javadoc
6. Interpreter/ loader

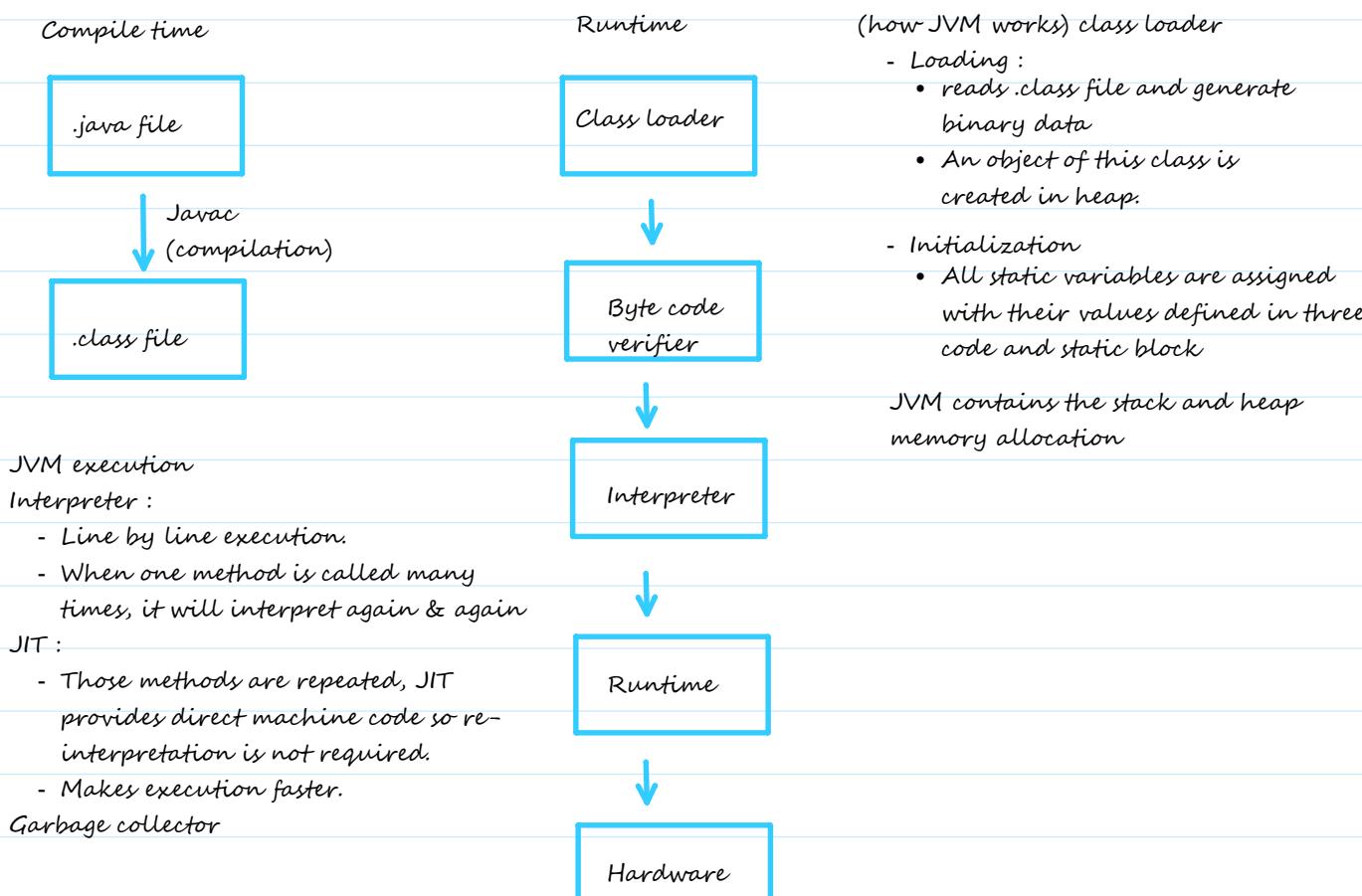
## ➤ JRE

- It is an installation that provides environment to only run the program.
- It consists of:

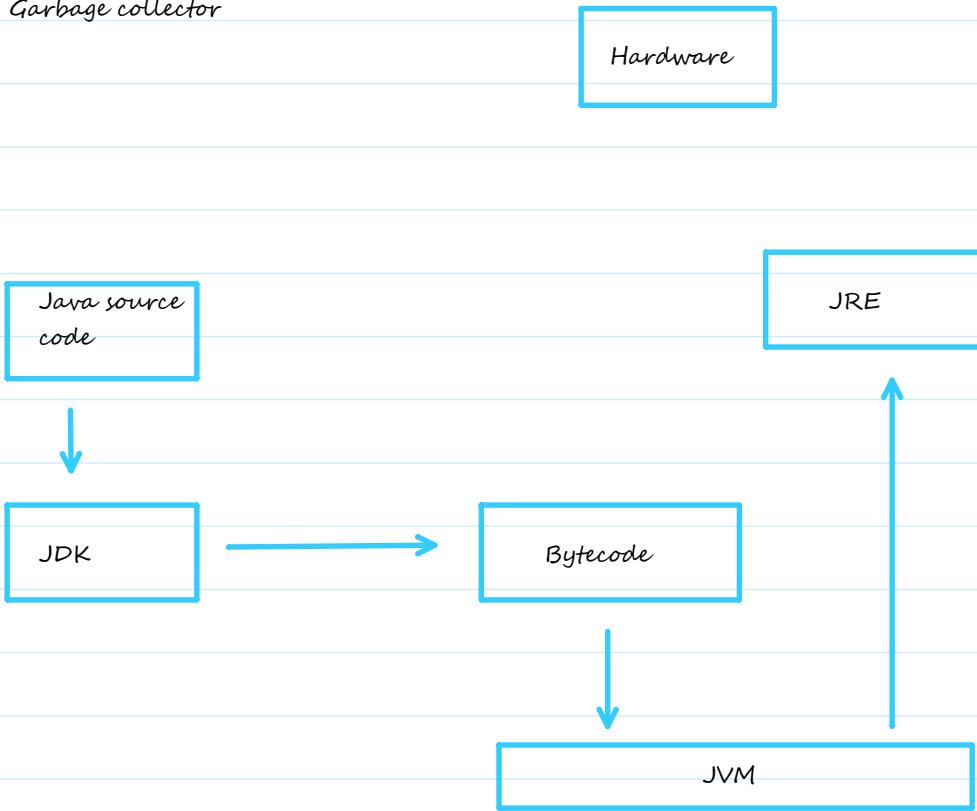
1. Deployment technologies
2. User interface toolkits
3. Integration libraries
4. Base libraries
5. JVM

- After we get the .class file, the next things happen at runtime:

1. Class loader loads all classes needed to execute the program.
2. JVM sends code to byte code verifier to check the format of code.



Garbage collector



## STRUCTURE OF JAVA PROGRAM

21 August 2021 09:54

```
package com.company;
import java.util.Scanner;
public class Inputs {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

//main function in java

- Make sure that whenever you will create class make first letter capital ~ convention (It's not like that if you will not write first letter capital it will not work...it will work but it's not a good convention)
- Main.java *It will compile into byte code* --> Main.class (byte code) *Interpreter* --> Machine code (0 and 1)
- If the name of the file is Demo than class name has to be demo
- Public means class or function can be accessed anywhere.
- Class is group of properties and function.
- Static is actually means when we want to run anything without creating object.  
 Here static we have used because we want to run main function without creating object of class demo.
- `String[] args` is actually command line argument. Whatever argument you are giving in the terminal as in the array it will stored in the terminal

```
package com.company;
import java.util.Scanner;
public class Inputs {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

- package `com.company` basically it is folder where you all java file lies



- `System.out.println("hello world");` //if you want new line or else  
`System.out.print("hello world");`
- Some shortcuts:
  - Psvm enter :

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

- Sout enter: System.out.println();

➤ Whenever we create 2 class in a file it will generate 2 byte code file

```
package com.roshan;
public class Demo {
    public static void main(String[] args) {
    }
}
class Demo2{
}
```

File name : Demo.java



----->

```
package com.roshan;
public class Demo {
    public static void main(String[] args) {
    }
}
Public class Demo2{
}
```

//error --> both class name can't be public  
There can be only one public class in a java file because  
the name of java file is same as the name of public class.  
And obviously we can't have a file with two different  
names

----->

```
package com.roshan;
class Demo {
    public static void main(String[] args) {
    }
}
class Demo2{
```

// it will work  
// here file name can be any thing

Java says if the class is public then the file  
name and class name has to be same otherwise  
you will get an error.

➤ Runtime error and compile time error

- **Compile-Time Errors:** Errors that occur when you violate the rules of writing syntax are known as Compile-Time errors. This compiler error indicates something that must be fixed before the code can be compiled. All these errors are detected by the compiler and thus are known as compile-time errors.

Most frequent Compile-Time errors are:

- Missing Parenthesis { }
- Printing the value of variable without declaring it
- Missing semicolon (terminator)

- **Run-Time Errors:** Errors which occur during program execution(run-time) after successful

compilation are called run-time errors. One of the most common run-time error is division by zero also known as Division error. These types of error are hard to find as the compiler doesn't point to the line at which the error occurs.

- Difference between compile time error and runtime error

### Compile-Time Errors

- These are the syntax errors which are detected by the compiler.
- They prevent the code from running as it detects some syntax errors.
- It includes syntax errors such as missing of semicolon(;), misspelling of keywords and identifiers etc.

### Runtime-Errors

- These are the errors which are not detected by the compiler and produce wrong results.
- They prevent the code from complete execution.
- It includes errors such as dividing a number by zero, finding square root of a negative number etc.

## ➤ Javac and java

The javac command is used to compile Java programs, it takes .java file as input and produces bytecode. Following is the syntax of this command.

>javac sample.java

The java command is used to execute the bytecode of java. It takes byte code as input and runs it and produces the output. Following is the syntax of this command.

>java sample

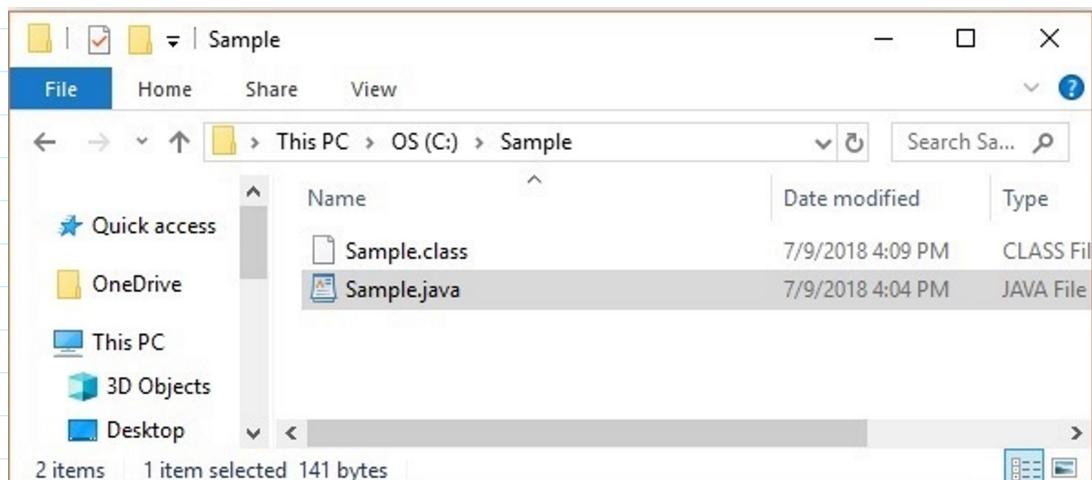
Let us consider an example create a Sample Java program with name Sample.java

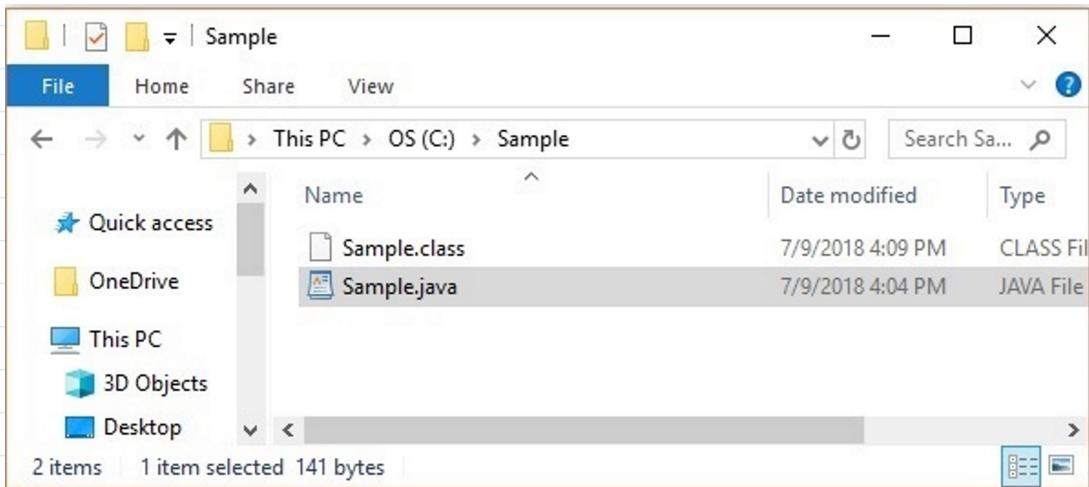
```
public class Sample {  
    public static void main(String args[]) {  
        System.out.println("Hi my name is roshan prusty");  
    }  
}
```

If you compile this program using Javac command as shown below -

C:\Examples >javac Sample.java //write javac filename.java

This command compiles the given java file and generates a .class file (byte code)





And if you execute the generated byte code (.class) file using the java command as shown below -

C:\Sample>java Sample //write java class name

Hi my name is roshan prusty

- How to Compile and Run Java Program from Command Prompt
  - Link : <https://www.youtube.com/watch?v=zBF1M8oTfHk>
- How to Compile and Run Java Program in Command Prompt ( With Packages )
  - Link : <https://www.youtube.com/watch?v=QWUFPehCjII>

# Java Command Line Arguments

05 February 2022 02:35

## ➤ What is Command Line Argument in Java?

Command Line Argument in Java is the information that is passed to the program when it is executed. The information passed is stored in the string array passed to the main() method and it is stored as a string. It is the information that directly follows the program's name on the command line when it is running.

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behaviour of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

## ➤ Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{  
public static void main(String args[]){  
System.out.println("Your first argument is: "+args[0]);  
}  
}
```

compile by > javac CommandLineExample.java  
run by > java CommandLineExample sonoo  
Output: Your first argument is: sonoo

## ➤ Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A{  
public static void main(String args[]){  
for(int i=0;i<args.length;i++)  
System.out.println(args[i]);  
}  
}
```

compile by > javac A.java  
run by > java A sonoo jaiswal 1 3 abc  
  
Output:  
sonoo  
jaiswal  
1  
3  
abc

- Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.
- There are two data types available in Java -
  - Primitive Data Types

- Variable of Primitive data type hold the value of the data item
- Primitive data type is basically a data type that you cannot break even further.
- Example int, char and etc
- String is not a primitive data type -
 

```
String name = "roshan" //here we can break roshan into character
```

- byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 ( $-2^7$ )
- Maximum value is 127 (inclusive) ( $2^7 - 1$ )
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

- short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 ( $-2^{15}$ )
- Maximum value is 32,767 (inclusive) ( $2^{15} - 1$ )
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0
- Example: short s = 10000, short r = -20000

- int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is -2,147,483,648 ( $-2^{31}$ )
- Maximum value is 2,147,483,647 (inclusive) ( $2^{31} - 1$ )
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

**Topic to add here BigInteger...**

**Integer..... --> explained in OOP section**

- Long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808 ( $-2^{63}$ )
- Maximum value is 9,223,372,036,854,775,807 (inclusive) ( $2^{63} - 1$ )
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

> Non-primitive : Variable of Non-primitive data type hold the reference of the memory location where the data object is stored.

These are also called as Reference data type.

//to get more accurate value that time we use double

# Decimal type data type

08 May 2022 02:09

- Real numbers measure continuous quantities, like weight, height, or speed. Floating point numbers represent an approximation of real numbers in computing. In Java we have two primitive floating point types: float and double. The float is a single precision type which store numbers in 32 bits. The double is a double precision type which store numbers in 64 bits. These two types have fixed precision and cannot represent exactly all real numbers. In situations where we have to work with precise numbers, we can use the BigDecimal class.

Floating point numbers with an F/f suffix are of type float, double numbers have D/d suffix. The suffix for double numbers is optional.

## - Float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

## - double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

## • The float and double data type Deep Dive in Java

- Don't use float and double on the financial calculation, instead prefer BigDecimal. If you have to use float, then limit the precision like 8 places up to the decimal point
- Don't compare float and double using == operator, Instead use > or <, this stands for the less than sign (>); or <, this stands for the less-than sign (<) you may be wondering why? Well, if you use == with float/double as loop terminating condition, then this may result in an endless loop because a float/double may not be precisely equal to sometimes as Java uses binary floating-point calculation, which results in approximation.

### BRIEF:

Why you shouldn't use == with float and double in Java?

Especially for checking loop termination conditions. Java programmers often make the mistake of using floating-point numbers in a loop and checking conditions with the == operator, in the worst case this could create an infinite loop, causing your Java application to hung.

For example, the following code will not work as you expect:

```
for(double balance=10; balance>0; balance-=0.1) {
    System.out.println(balance);
}
```

You would think that this code will print the balance until the balance reduced to zero. Since we are doing  $balance = balance - 0.1$ , you would expect it to print something like 10, 9.9, 9.8, and so on until it reaches zero.

But to your surprise, this will cause an infinite loop in Java, it will never end. Why? because 0.1 is an infinite binary decimal, the balance will never be exactly 0.

This is the reason many programmers, including Joshua Bloch, have suggested avoiding using double and float for monetary calculation in his book Java Puzzlers.

Now, this brings a more pertinent question, how do you compare floating point numbers in Java? if == is not going to work then how can I write a similar loop, because that's a very general case and it would be unfortunate if you can't test for decimal points? Let's find out the right way to compare floating-point values in Java in the next section.

#### How to compare float and double values in Java?

As we have seen in the first paragraph that use of == operator can cause an endless loop in Java, but is there a way to prevent that loop from running infinitely? Yes, instead of using equality operator (==), you can use relational operator e.g. less than (<) or greater than (>) to compare float and double values.

By changing the above code as following, you can prevent the loop from infinitely :

```
for(double balance=10; balance>0; balance-=0.1) {
    System.out.println(balance);
}
```

If you think a little bit then you will realize that greater than will definitely end this loop, but don't expect it to print numbers like 9.9, 9.8, 9.7 because floating-point numbers are a just approximation, they are not exact. Some numbers like  $1/3$  cannot be represented exactly using float and double in Java.

After running the following program on your computer you may end up with something like this

```
/**
 * Don't use == operator with float and double values in Java
 *
 * @author Javin Paul
 */
public class FloatInForLoop {
    public static void main(String args[]) {
        for (double balance = 10; balance > 0; balance -= 0.1) {
            System.out.println(balance);
        }
    }
}
```

```
}
```

Output:

```
10.0  
9.9  
9.8  
9.70000000000001  
9.60000000000001  
9.50000000000002  
9.40000000000002  
9.30000000000002  
...  
....  
0.900000000000187  
0.800000000000187  
0.700000000000187  
0.600000000000187  
0.500000000000188  
0.400000000000188  
0.300000000000188  
0.200000000000188  
0.100000000000188  
1.8790524691780774E-14
```

You can see that result is nowhere close to our expectation, and that's why float and double are not recommended for financial calculation or where the exact result is expected.

#### Some tips while using float and double in Java

Do all calculations in float/double but for comparison, always compare approximation instead of precise values e.g. instead of checking for 10.00 check for > 9.95 as shown below

```
if(amount ==100.00) // Not Ok  
if(amount > 99.995) // Ok
```

One reason, why many programmers make the mistake of comparing floating points number with == operator is that for some fractions, it does work properly like if we change 0.1 to 0.5 the loop in question will work properly as shown below:

```
for(doublebalance=10; balance>0; balance-=.5){  
    System.out.println(balance);  
}  
Output:  
10.0  
9.5  
9.0  
8.5  
8.0  
7.5
```

7.0  
6.5  
6.0  
5.5  
5.0  
4.5  
4.0  
3.5  
3.0  
2.5  
2.0  
1.5  
1.0  
0.5

Alternatively, you can use BigDecimal for exact calculation. You can also try rewriting the same code using BigDecimal class instead of double primitive.

That's all about why you shouldn't use == operator with float and double in Java. If you have to check conditions involving float and double values then instead of using == always use relational operator e.g. < or > for comparing floating-point numbers in Java.

I repeat, don't use double and float for monetary calculation unless you are an expert of floating-point numbers, know about precision, and have a very good understanding of floating-point arithmetic in Java.

3. While using float and double in hashCode() method, use them as long, as suggested in Effective Java, for example Double.longBits() or Float.longBits()
  
4. The program often confuses between the maximum and minimum values of double and float, unlike int and long, they cannot be calculated based upon the size of float and double in bits, like 32 and 64. That's why the maximum value of long can be stored in float without any casting, but you cannot store a float value in long without casting, as displayed in the following example:

```
double maxDouble = Double.MAX_VALUE;  
double minDouble = Double.MIN_VALUE;  
float maxFloat = Float.MAX_VALUE;  
float minFloat = Float.MIN_VALUE;
```

```
System.out.println("Maximum value of double data type in Java : " + maxDouble);  
System.out.println("Minimum value of double in Java : " + minDouble);  
System.out.println("Maximum value of float data type in Java : " + maxFloat);  
System.out.println("Minimum value of float in Java : " + minFloat);
```

```
float myfloat = Long.MAX_VALUE; //OK  
long myLong = Float.MAX_VALUE; // Not Ok, Type mismatch, cannot convert from float to long
```

Output:

The maximum value of double data type in Java: 1.7976931348623157E308

The minimum value of double in Java: 4.9E-324

The maximum value of float data type in Java: 3.4028235E38

The minimum value of float in Java: 1.4E-45

5. You can represent infinity using float/double in Java, which may come as a surprise to you, but it's true. Both Float and Double wrapper classes have constant values to represent infinity as

```
Double.POSITIVE_INFINITY,  
Double.NEGATIVE_INFINITY,  
Float.POSITIVE_INFINITY,  
Float.NEGATIVE_INFINITY.
```

```
double a = Double.POSITIVE_INFINITY;  
double b = Double.NEGATIVE_INFINITY;  
float c = Float.POSITIVE_INFINITY;  
float d = Float.NEGATIVE_INFINITY;  
  
System.out.println(a); //Infinity  
System.out.println(b); // -Infinity  
System.out.println(c); //Infinity  
System.out.println(d); // -Infinity
```

They also have a method to test, whether a double/float value is infinity or not, by using `isInfinite()` method, as shown in the following example:

```
double infinity = 1.0/0.0; // divide by zero is infinity in maths  
System.out.println(Double.isInfinite(infinity)); // prints true
```

Remember adding a small floating-point value will not change the large floating-point value, which means adding something on infinity will also be infinity, as shown below:

```
double infinity = 1.0/0.0; // divide by zero is infinity in maths  
infinity = infinity + 100;  
System.out.println(Double.isInfinite(infinity)); // still true
```

#### - Big decimal

`BigDecimal` is used to represent bigger numbers in Java, similar to `float` and `long` but many Java programmers also use `BigDecimal` to store floating-point values because it's easier to perform arithmetic operations on `BigDecimal` than `float` or `double`. That's the reason, many Java programmer uses `BigDecimal` for monetary or financial calculation that floats or double.

In this example, we have used `BigDecimal` inside a for loop and also demonstrate how to perform arithmetic and logical operation with `BigDecimals` like addition, subtraction, comparison, etc.

```

BigDecimal TEN_DOLLARS = new BigDecimal("10.0");
BigDecimal TEN_CENTS = new BigDecimal("0.1");
//AlwaysUse String constructor and not double one

BigDecimal ZERO_CENTS = new BigDecimal("0.0");

for(BigDecimal balance= TEN_DOLLARS; balance.compareTo(ZERO_CENTS) >= 0 ;
balance = balance.subtract(TEN_CENTS)){
    System.out.println(balance);
}

```

Output:

```

10.0
9.9
9.8
0.2
0.1
0.0

```

### More BigDecimal Examples in Java

That was a really simple example of using BigDecimal inside a for loop and decreasing value until it becomes zero. Now let's see few more BigDecimal examples about addition, subtraction, and multiplication of BigDecimal in Java.

```

package com.roshan;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class Decimal_value {

    public static void main(String args[]) {

        // Creating instance of BigDecimal in Java
        BigDecimal twentyOne = new BigDecimal("21.01");
        // String constructor behaves as they seen

        BigDecimal twentyTwo = new BigDecimal("22.01");
        BigDecimal _22 = new BigDecimal(22.01);
        // double constructor, doesn't behave as they look

        System.out.println(twentyOne + ", " + twentyTwo + ", " + _22);

        // How to get the scale of BigDecimal number
        int scale = twentyOne.scale();
        System.out.printf("scale of BigDecimal %s is %d %n", twentyOne, scale);
        int _scale = _22.scale();
        System.out.printf("scale of BigDecimal %s is %d %n", _22, _scale);
    }
}

```

```
//How to set the scale of BigDecimal number  
_22 = _22.setScale(4, RoundingMode.HALF_DOWN);  
// may return a different BigDecimal object  
System.out.printf("new scale of BigDecimal %s is %d %n", _22, _22.scale());
```

```
// Adding two BigDecimal numbers in Java  
BigDecimal sum = twentyOne.add(twentyOne); // adding BigDecimal of same scale  
BigDecimal addition = twentyOne.add(_22); // adding BigDecimal of different scale  
System.out.printf("sum of %s and %s is %s %n", twentyOne, twentyTwo, sum);  
System.out.printf("addition of %s and %s is %s %n", twentyOne, _22, addition);
```

```
// Subtracting two BigDecimal in Java  
BigDecimal minus = twentyOne.subtract(twentyOne);  
// subtracting BigDecimal of same scale  
BigDecimal subtraction = twentyOne.subtract(_22);  
// subtracting BigDecimal of different scale  
System.out.printf("minus of %s and %s is %s %n", twentyOne, twentyTwo, minus);  
System.out.printf("subtraction of %s and %s is %s %n", twentyOne, _22,  
subtraction);
```

```
// Multiplying BigDecimals in Java  
BigDecimal product = twentyOne.multiply(twentyOne);  
// multiplying BigDecimal of same scale  
  
BigDecimal multiplication = twentyOne.multiply(_22);  
// multiplying BigDecimal of different scale  
System.out.printf("product of %s and %s is %s %n", twentyOne,  
twentyTwo, product);  
System.out.printf("multiplication of %s and %s is %s %n",  
twentyOne, _22, multiplication);
```

```
// Divide two BigDecimal in Java  
BigDecimal two = new BigDecimal("2.0");  
BigDecimal divide = twentyOne.divide(two); // dividing BigDecimal of same scale  
BigDecimal division = _22.divide(two); // dividing BigDecimal of different scale  
System.out.printf("divide of %s and %s is %s %n", twentyOne, two, divide);  
System.out.printf("division of %s and %s is %s %n", _22, two, division);  
}  
}
```

#### Output

```
21.01, 22.01, 22.010000000000001563194018672220408916473388671875  
scale of BigDecimal 21.01 is 2  
scale of BigDecimal 22.010000000000001563194018672220408916473388671875 is 48  
new scale of BigDecimal 22.0100 is 4  
sum of 21.01 and 22.01 is 42.02  
addition of 21.01 and 22.0100 is 43.0200  
minus of 21.01 and 22.01 is 0.00  
subtraction of 21.01 and 22.0100 is -1.0000  
product of 21.01 and 22.01 is 441.4201
```

multiplication of 21.01 and 22.0100 is 462.430100

divide of 21.01 and 2.0 is 10.505

division of 22.0100 and 2.0 is 11.005

### Things to note while using BigDecimal

Here are some important details about BigDecimal class and values in Java. Knowing these details is mandatory for effective using BigDecimal in a real-world Java application.

#### 1. Prefer String Constructor

You should always use the String constructor of BigDecimal i.e. new BigDecimal("20.0"), rather than a double constructor like new BigDecimal(20.0), though they look very similar, there is a big difference between them. The results of this constructor can be unpredictable.

You may assume that creating new BigDecimal(0.1) in Java creates a BigDecimal which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000055511151231257827021181583404541015625.

This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed into the constructor is not exactly equal to 0.1, appearances notwithstanding.

#### 2. Same Scale

Always operate with BigDecimal of the same scale, otherwise, the result would be unpredictable. For example in our case, all BigDecimal number has only one number after the decimal point. You would notice that I have not used BigDecimal.TEN or BigDecimal.ZERO constants from BigDecimal class itself because they have different scales and using them will result in unpredictable results similar to using double values.

#### 3. Immutable

BigDecimal is an immutable class in Java. It also caches frequently used values like boxed primitive classes e.g. Integer and Long.

#### 4. Internal Representation

BigDecimal internally uses an int and int array to represent numbers. When you create another BigDecimal number, which just differs in sign e.g. by using negate() method then these two BigDecimal share the same int array.

5. BigDecimal class also has a companion mutable class to reduce temporary objects, which is created during a multi-step calculation process. Read Effective Java for more details.

6. float and double data types are mainly provided for scientific and engineering calculations. Java uses binary floating-point calculations which are good for approximation but don't provide exact results. The bottom line is, don't use float/double when an exact calculation is needed. You cannot represent values like 0.1 or 0.01 or any negative power of 10 accurately in Java. calculating interest, expenses is one example of this.

7. compareTo() and equals() method of java.lang.BigDecimal class is inconsistent with each

other. This means comparing "3.0" and "3.00" with equals() and compareTo() will give a different result, the former will return false, while later will return true. Why? because equals() method of BigDecimal only return true if both numbers are the same in value and scale, while the compareTo() method only compares values.

This inconsistent behavior of BigDecimal can leads to subtle issues when used with Collection classes, which make use of both equals() and compareTo() method e.g. if you put both "3.0" and "3.00" in a Set, it will allow both of them, which is wrong, given Set doesn't allow duplicates, as shown in below example :

```
BigDecimal THREE = new BigDecimal("3.0");
BigDecimal three = new BigDecimal("3.00");
```

```
Set biggies = new HashSet();
biggies.add(THREE);
biggies.add(three);
```

```
System.out.println("Size: " + biggies.size());
System.out.println(biggies);
```

```
System.out.println("Does biggies contains 3.0 : " + biggies.contains(new BigDecimal("3.0")));
System.out.println("Does biggies contains 3.000 : " + biggies.contains(new BigDecimal("3.000")));
```

Output:

```
Size: 2[3.00, 3.0]
Does biggies contains 3.0: true
Does biggies contains 3.000: false
```

You can see that HashSet allowed both "3.0" and "3.00", which may look like violation of its contract of not allowing duplicate, but it does because both of these BigDecimal number is not equal by equals() method. Similarly contains() method false for "3.000" because it also uses equals() method for checking if an object exists inside Set or not.

Now let's change the Set implementation from HashSet to TreeSet, which uses compareTo() method for comparison, here is how our code and result look like :

```
BigDecimal THREE = new BigDecimal("3.0");
BigDecimal three = new BigDecimal("3.00");
```

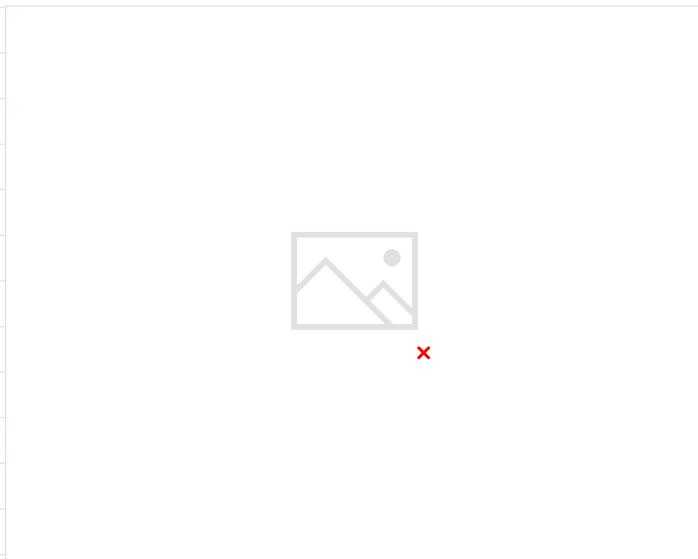
```
Set treeSet = new TreeSet();
treeSet.add(THREE);
treeSet.add(three);
```

```
System.out.println("Size: " + treeSet.size());
System.out.println(treeSet);
```

```
System.out.println("Does TreeSet contains 3.0 : "
+ treeSet.contains(new BigDecimal("3.0")));
System.out.println("Does TreeSet contains 3.000 : "
+ treeSet.contains(new BigDecimal("3.000")));
Size: 1
[3.0]
Does TreeSet contains 3.0 : true
```

Does TreeSet contains 3.000 : true

Now it seems like TreeSet is following Set contract and not allowing duplicate, while HashSet is not, but the truth is they both are using different methods for comparison, TreeSet is using compareTo() while HashSet is using equals() and because they are inconsistent to each other, we are getting different behavior by different Set implementations.



### Lesson learned :

1. Pay attention to both scale and value while comparing two BigDecimal or checking for equality. Choose equals() if you want to consider both value and scale and choose compareTo() if two BigDecimal numbers can be equal by just values.
2. While overriding compareTo() and equals() method for a class in Java, make sure they are consistent with each other.
3. Be careful while storing BigDecimal in Collection classes e.g. Set and Map especially while using them as key in SortedMap and storing them as inside SortedSet because BigDecimal's natural ordering is inconsistent with equals.

That's all about BigDecimal in Java. It's one of the important class, which is not well understood by many Java developers. I suggest programmers of all level to develop a good understanding of java.lang.BigDecimal to avoid surprises and bugs, when you really need to use it. Almost half of the bugs, which involves BigDecimal is because of poor understanding of this class and its behaviour. Programmer face issues when using float/double and blindly turns to BigDecimal, without knowing about scale, value and which BigDecimal constructor to use for which purpose.

4. Scale of a BigDecimal is a number of digit after decimal point, for example scale of BigDecimal 10.1 is 1, 10.01 is 2 and 10.001 is 3. You can get scale of a BigDecimal by calling scale() method and you can also change scale by calling setScale() method. Remember, since BigDecimal is Immutable, setScale may return a new BigDecimal object,

of course only if new scale is different than old scale. By the way you can also specify RoundingMode while changing scale, BigDecimal provides overloaded version of setScale() method. Prefer to new setScale(int, RoundingMode) over legacy setScale(int, int), which also provide same functionality. Remember, setScale(int) method may throw Exception in thread "main" java.lang.ArithmeticException: Rounding necessary, if it found that Rounding is needed.

5. BigDecimal is a sub-class of Number, which means you can pass a BigDecimal object to any method, which is accepting Number object. Similarly a method can return a BigDecimal object if its return type is Number. This makes BigDecimal a close cousin of float/double and other numeric primitive types.
6. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value of the number is multiplied by ten to the power of the negation of the scale. The value of the number represented by the BigDecimal is therefore (unscaled Value  $\times$  10 $^{-}$ scale). add few more points from javadoc
7. Immutable, arbitrary-precision signed decimal numbers. A BigDecimal consists of an arbitrary precision integer unscaled value and a 32-bit integer scale
8. Using BigDecimal has the added advantage that it gives you full control over rounding, letting you select from eight rounding modes whenever an operation that entails rounding is performed. This comes in handy if you're performing business calculations with legally mandated rounding behaviour. If performance is of the essence, you don't mind keeping track of the decimal point yourself, and the quantities aren't too big, use int or long. If the quantities don't exceed nine decimal digits, you can use int; if they don't exceed eighteen digits, you can use long. If the quantities might exceed eighteen digits, you must use BigDecimal.
9. It was not widely understood that immutable classes had to be effectively final when BigInteger and BigDecimal were written, so all of their methods may be overridden. Unfortunately, this could not be corrected after the fact while preserving backward compatibility. If you write a class whose security depends on the immutability of a BigInteger or BigDecimal argument from an untrusted client, you must check to see that the argument is a "real" BigInteger or BigDecimal, rather than an instance of an untrusted subclass. If it is the latter, you must defensively copy it under the assumption that it might be mutable (Item 39):

```
public static BigInteger safeInstance(BigInteger val) {  
    if (val.getClass() != BigInteger.class)  
        return new BigInteger(val.toByteArray());  
    return val;  
}
```

That's all about how to use BigDecimal in Java. We have not only seen how to use BigDecimal but also learned important details and best practices you can follow while using BigDecimal in Java like using String constructor and using the same scale while

*performing operations on BigDecimal.*

# Other data type

09 May 2022 02:01

## - boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

## - char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

# Primitive vs Non-primitive data type

30 June 2022 02:40

## Variable

28 January 2022 09:20

- A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

- **example**

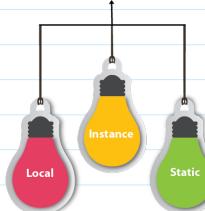
- `int a, b, c;` // Declares three ints, a, b, and c.
- `int a = 10, b = 10;` // Example of initialization.
- `byte B = 22;` // initializes a byte type variable B.
- `double pi = 3.14159;` // declares and assigns a value of PI.
- `char a = 'a';` // the char variable a is initialized with value 'a'

- This chapter will explain various variable types available in Java Language.

There are three kinds of variables in Java -

- Local variables
- Instance variables
- Class/Static variables

### Types of Variables



- **Local Variables**

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

- **Instance Variable**

- A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.
- It is called an instance variable because its value is instance-specific and is not shared among instances.

- **Static variable**

- A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

### Example to understand the types of variables in java

```
public class A
{
    static int m=100; //static variable
    void method()
    {
        int n=90; //local variable
    }
    public static void main(String args[])
    {
        int data=50; //instance variable
    }
}//end of class
```

### Java Variable Example: Add Two Numbers

```
public class Simple{
```

```

public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}
}
Output:
20

```

#### Java Variable Example: Widening

```

public class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
Output:
10
10.0

```

#### Java Variable Example: Narrowing (Typecasting)

```

public class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f; //Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
Output:
10.5
10

```

#### Java Variable Example: Overflow

```

class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
Output:
130
-126

```

#### Java Variable Example: Adding Lower Type

```

class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b; //Compile Time Error:
//because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);
}
}
Output:
20

```

 //error -- The addition of two-byte values in java is the same as normal integer addition. The byte data type is 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The sum of two-byte values might cross the given limit of byte, this condition can be handled using storing the sum of two-byte numbers in an integer variable.

Way 1:

```

byte a=2;
byte b=5;
int c = a + b;

```

Way 2:

```

byte a=2;
byte b=5;
byte c = (byte) (a + b)

```

# Inputs in java

05 January 2022 15:04

- Inputs in java

Scanner input = new Scanner(system.in)



system.in is basically default system input.  
You can also read from any where also like from file then cmd will be:  
Scanner input = new Scanner(file)

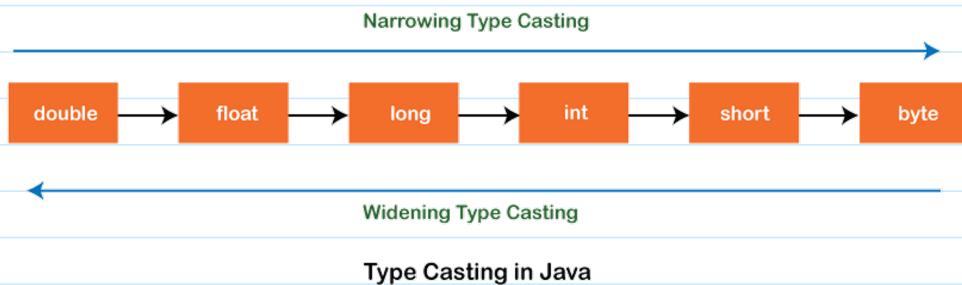
- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings
- Scanner is basically a class that specifies an input stream and using object of the class we can take input. Its in java.util package

```
package com.company;
import java.util.Scanner;
public class Inputs {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("enter any num ");
        int rollno = input.nextInt();
        System.out.println("your num is " + rollno);
    }
}
```

# Type casting

05 January 2022 16:23

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss type casting and its types with proper examples.



## Type casting

Convert a value from one data type to another data type is known as type casting.

## Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

## Widening Type Casting

Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

`byte -> short -> char -> int -> long -> float -> double`

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

### WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
    public static void main(String[] args)
    {
        int x = 7;

        //automatically converts the integer type into long type
        long y = x;
```

```
//automatically converts the long type into float type  
float z = y;
```

```
System.out.println("Before conversion, int value "+x);  
System.out.println("After conversion, long value "+y);  
System.out.println("After conversion, float value "+z);  
}  
}  
}
```

Output

Before conversion, the value is: 7

After conversion, the long value is: 7

After conversion, the float value is: 7.0

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

### Narrowing Type Casting

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample  
{  
    public static void main(String args[])  
    {  
        double d = 166.66;  
  
        //converting double data type into long data type  
        long l = (long)d;
```

```
        //converting long data type into int data type  
        int i = (int)l;
```

```
        System.out.println("Before conversion: "+d);
```

```
        //fractional part lost
```

```
        System.out.println("After conversion into long type: "+l);
```

```
        //fractional part lost
```

```
        System.out.println("After conversion into int type: "+i);  
    }  
}
```

Output

Before conversion: 166.66

After conversion into long type: 166

After conversion into int type: 166

# Decision making

16 January 2022 14:34

## Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

- Less than:  $a < b$
- Less than or equal to:  $a \leq b$
- Greater than:  $a > b$
- Greater than or equal to:  $a \geq b$
- Equal to:  $a == b$
- Not Equal to:  $a != b$

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

### ➤ The if Statement

Use the `if` statement to specify a block of Java code to be executed if a condition is true.

#### Syntax

```
if(condition){  
    // block of code to be executed if the condition is true  
}
```

Note that `if` is in lowercase letters. Uppercase letters (`If` or `IF`) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is true, print some text.

#### Example

```
if(20>18){  
    System.out.println("20 is greater than 18");  
}
```

We can also test variables:

#### Example

```
int x = 20;
```

```
int y =18;  
if(x >y){  
    System.out.println("x is greater than y");  
}
```

### Example explained

In the example above we use two variables, *x* and *y*, to test whether *x* is greater than *y* (using the *>* operator). As *x* is 20, and *y* is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

### ➤ The else Statement

Use the *else* statement to specify a block of code to be executed if the condition is false.

#### Syntax

```
if(condition){  
    // block of code to be executed if the condition is true  
}else{  
    // block of code to be executed if the condition is false  
}
```

### Example

```
int time =20;  
if(time <18){  
    System.out.println("Good day.");  
}else{  
    System.out.println("Good evening.");  
}  
// Outputs "Good evening."
```

### Example explained

In the example above, *time* (20) is greater than 18, so the condition is false. Because of this, we move on to the *else* condition and print to the screen "Good evening". If the *time* was less than 18, the program would print "Good day".

### ➤ The else if Statement

Use the *else if* statement to specify a new condition if the first condition is false.

#### Syntax

```
if(condition1){  
    // block of code to be executed if condition1 is true  
}elseif(condition2){  
    // block of code to be executed if the condition1 is false and condition2 is true  
}else{  
    // block of code to be executed if the condition1 is false and condition2 is false  
}
```

## Example

```
int time =22;  
if(time <10){  
    System.out.println("Good morning.");  
}elseif(time <20){  
    System.out.println("Good day.");  
}else{  
    System.out.println("Good evening.");  
}// Outputs "Good evening."
```

## Example explained

In the example above, time (22) is greater than 10, so the first condition is false. The next condition, in the else if statement, is also false, so we move on to the else condition since condition1 and condition2 is both false - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

## Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the ternary operator because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements.

## Syntax

```
variable =(condition)?expressionTrue :expressionFalse;  
Instead of writing:
```

## Example

```
int time =20;  
if(time <18){  
    System.out.println("Good day.");  
}else{  
    System.out.println("Good evening.");  
}
```

You can simply write:

## Example

```
int time =20;  
String result = (time <18) ? "Good day." : "Good evening." ;  
System.out.println(result);
```

➤ **Java Switch statement**

The switch statement allows us to execute a block of code among many alternatives.

The syntax of the switch statement in Java is:

```
switch(expression) {  
    case value1:  
        // code  
        break;  
  
    case value2:  
        // code  
        break;  
  
    ...  
    ...  
  
    default:  
        // default statements  
}
```

How does the switch-case statement work?

The expression is evaluated once and compared with the values of each case.

- If expression matches with value1, the code of case value1 are executed.  
Similarly, the code of case value2 is executed if expression matches with value2.
- If there is no match, the code of the default case is executed.

Example

```
// Java Program to check the size  
// using the switch...case statement  
  
class Main {  
    public static void main(String[] args) {  
  
        int number = 44;  
        String size;  
  
        // switch statement to check size  
        switch (number) {  
  
            case 29:  
                size = "Small";
```

```
break;  
  
case 42:  
    size = "Medium";  
    break;  
  
// match the value of week  
case 44:  
    size = "Large";  
    break;  
  
case 48:  
    size = "Extra Large";  
    break;  
  
default:  
    size = "Unknown";  
    break;  
  
}  
System.out.println("Size: " + size);  
}  
}
```

Output: size = large

# loop

16 January 2022 14:34

**Loops:** Loops are used to iterate a part of program several times.

1. **For Loop:-** It is generally used when we know how many times loop will iterate.

Syntax :-

```
for(init; condition; incr/decr){  
    // code to be executed  
}
```

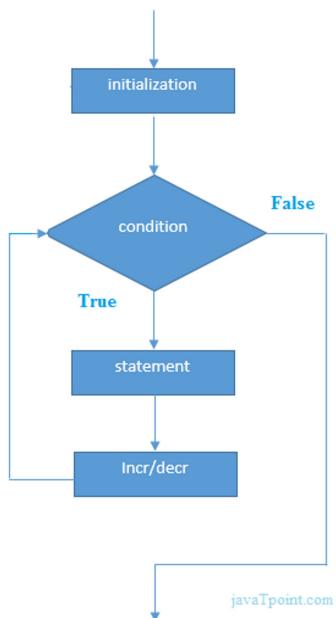
**Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

**Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return Boolean value either true or false. It is an optional condition.

**Increment/Decrement:** It increments or decrements the variable value. It is an optional condition.

**Statement:** The statement of the loop is executed each time until the second condition is false.

**Flow chart**



**Example**

```
//Java Program to demonstrate the example of for loop  
//which 1 to 5
```

```
public class ForExample {  
    public static void main(String[] args) {  
        //Code of Java for loop  
        for(int i=1;i<=5;i++){  
            System.out.println(i);  
        }  
    }  
}
```

*Output:* 1

2  
3  
4  
5

- Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

*Example:*

*NestedForExample.java*

```
public class NestedForExample {  
    public static void main(String[] args) {  
        //loop of i  
        for(int i=1;i<=3;i++){  
            //loop of j  
            for(int j=1;j<=3;j++){  
                System.out.println(i+" "+j);  
            } //end of j  
        } //end of i  
    }  
}
```

*Output:*

1 1  
1 2  
1 3  
2 1  
2 2  
2 3  
3 1  
3 2  
3 3

### Pyramid example

#### Pyramid Example 1: *PyramidExample.java*

```
public class PyramidExample {  
    public static void main(String[] args) {  
        for(int i=1;i<=5;i++){  
            for(int j=1;j<=i;j++){  
                System.out.print("* ");  
            }  
            System.out.println(); //new line  
        }  
    }  
}
```

*Output:*

\*  
\* \*  
\* \* \*  
\* \* \* \*

#### Pyramid Example 2: *PyramidExample2.java*

```
public class PyramidExample2 {  
    public static void main(String[] args) {  
        int term=6;  
        for(int i=1;i<=term;i++){  
            for(int j=term;j>=i;j--){  
                System.out.print("* ");  
            }  
            System.out.println(); //new line  
        }  
    }  
}
```

*Output:*

\* \* \* \* \*  
\* \* \* \* \*  
\* \* \* \* \*

\* \*  
\* \* \*  
\* \* \* \*  
\* \* \* \* \*

\* \* \* \*  
\* \* \* \*  
\* \* \*  
\* \*  
\*

### • Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name){  
    //code to be executed  
}
```

Example:

ForEachExample.java

```
//Java For-each loop example which prints the  
//elements of the array  
public class ForEachExample {  
    public static void main(String[] args) {  
        //Declaring an array  
        int arr[]={12,23,44,56,78};  
        //Printing array using for-each loop  
        for(int i:arr){  
            System.out.println(i);  
        }  
    }  
}
```

Output:

```
12  
23  
44  
56  
78
```

### • Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

Note: The break and continue keywords breaks or continues the innermost for loop respectively.

Syntax:

```
labelname:  
for(initialization; condition; increment/decrement){  
    //code to be executed  
}
```

### For-Each Loop for 2D Arrays

```
String[][]array; // Nested For-each loops that traverse a 2D String array  
for(String[] innerArray : array){  
    for(String val : innerArray){  
        System.out.println(val);  
    }  
}
```

Example :

```
boolean[][] board = {  
    {true, true, true, false},  
    {true, true, true, true},  
    {true, true, true, true}  
};  
for (boolean[] arr : board){  
    for (boolean check:arr){  
        System.out.print(check+" ");  
    }  
    System.out.println();  
}
```

```
true true true false  
true true true true  
true true true true
```

**Example:**

**LabeledForExample.java**

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
    public static void main(String[] args) {
        //Using Label for outer and for loop
        aa:
        for(int i=1;i<=3;i++){
            bb:
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    break aa;
                }
                System.out.println(i+" "+j);
            }
        }
    }
}
```

**Output:**

```
1 1
1 2
1 3
2 1
```

If you use break bbs, it will break inner loop only which is the default behaviour of any loop.

**LabeledForExample2.java**

```
public class LabeledForExample2 {
    public static void main(String[] args) {
        aa:
        for(int i=1;i<=3;i++){
            bb:
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    break bb;
                }
                System.out.println(i+" "+j);
            }
        }
    }
}
```

**Output:**

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

- **Java Infinitive for Loop**

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

**Syntax:**

```

for(;;){
    //code to be executed
}

```

**Example:**  
ForExample.java

```

//Java program to demonstrate the use of infinite for loop
//which prints an statement

```

```

public class ForExample {
    public static void main(String[] args) {
        //Using no condition in for loop
        for(;;){
            System.out.println("infinitive loop");
        }
    }
}

```

**Output:**

```

infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c

```

Now, we need to press ctrl+c to exit from the program.

- Java for Loop vs while Loop vs do-while Loop

Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given Boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given Boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	for(init;condition;i ncr/decr){ // code to be executed }	while(condition){ //code to be executed }	do{ //code to be executed }while(condition);
Example	//for loop for(int i=1;i<=10;i++){ System.out.println(i); }	//while loop int i=1; while(i<=10){ System.out.println(i);     i++; }	//do-while loop int i=1; do{ System.out.println(i);     i++; }while(i<=10);
Syntax	for(;;){ //code to be executed }	while(true){ //code to be executed }	do{ //code to be executed }while(true);

ive  
loop

}

# While loop

08 February 2022 00:53

## Java While Loop

- The Java while loop is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.
- The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

```
while (condition){  
    //code to be executed  
    I ncrement / decrement statement  
}
```

## The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

Example:

```
i <=100
```

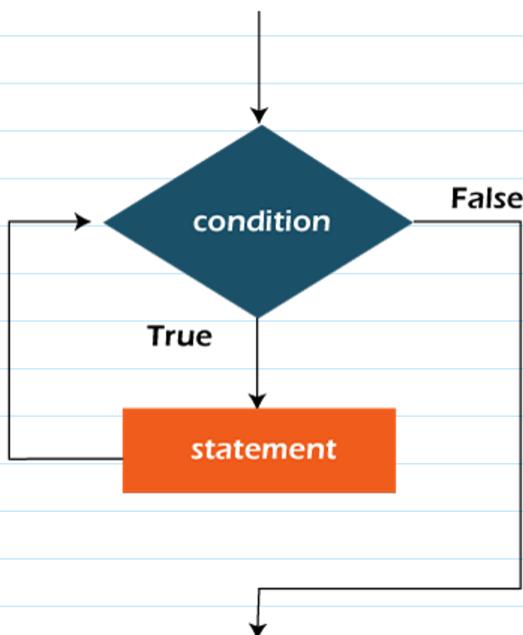
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

Example:

```
i++;
```

## Flowchart of Java While Loop

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



### Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, *i*). Otherwise, the loop will execute infinitely.

### WhileExample.java

```

public class WhileExample {
    public static void main(String[] args) {
        int i=1;
        while(i<=10){
            System.out.println(i);
            i++;
        }
    }
}

```

### Output:

```

1
2
3
4
5
6
7
8
9
10

```

### Java Infinitive While Loop

If you pass true in the while loop, it will be infinitive while loop.

**Syntax:**

```
while(true){  
//code to be executed  
}
```

**Example:**

```
WhileExample2.java  
public class WhileExample2 {  
public static void main(String[] args) {  
// setting the infinite while loop by passing true to the condition  
while(true){  
    System.out.println("infinitive while loop");  
}  
}  
}
```

**Output:**

```
infinitive while loop  
ctrl+c
```

In the above code, we need to enter *Ctrl + C* command to terminate the infinite loop.

# do-while loop

08 February 2022 00:53

## Java do-while Loop

The Java do-while loop is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an exit control loop. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

```
do{  
//code to be executed / loop body  
//update statement  
}while (condition);
```

### The different parts of do-while Loop:

1. **Condition:** It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

Example:

```
i <=100
```

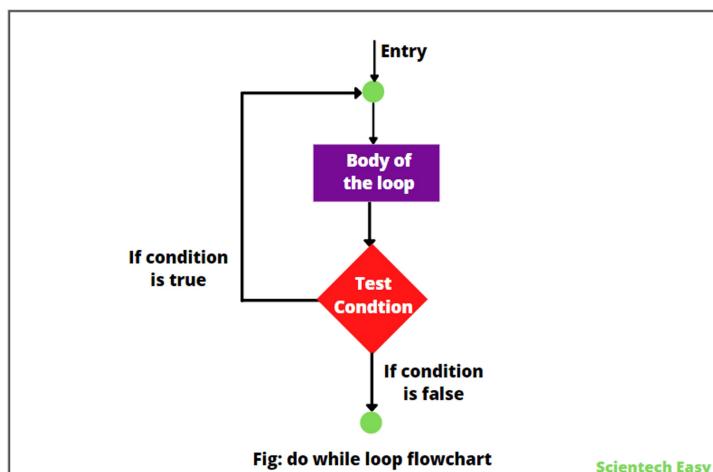
2. **Update expression:** Every time the loop body is executed, the this expression increments or decrements loop variable.

Example:

```
i++;
```

Note: The do block is executed at least once, even if the condition is false.

### Flowchart of do-while loop:



Example:

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

DoWhileExample.java

```
public class DoWhileExample {  
    public static void main(String[] args) {  
        int i=1;  
        do{  
            System.out.println(i);  
            i++;  
        }while(i<=10);  
    }  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

### Java Infinitive do-while Loop

If you pass true in the do-while loop, it will be infinitive do-while loop.

Syntax:

```
do{  
    //code to be executed  
}while(true);
```

Example:

DoWhileExample2.java

```
public class DoWhileExample2 {  
    public static void main(String[] args) {  
        do{  
            System.out.println("infinitive do while loop");  
        }while(true);  
    }  
}
```

Output:

infinitive do while loop

infinitive do while loop

infinitive do while loop

ctrl+c

In the above code, we need to enter Ctrl + C command to terminate the infinite loop.

## Function

14 January 2022 21:31

- A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println()` method, for example, the system actually executes several statements in order to display a message on the console.
- Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

### Creating Method

Considering the following example to explain the syntax of a method -

#### Syntax

```
public static int methodName(int a, int b){  
    // body  
}
```

Here,

- `public static` - modifier
- `int` - return type
- `methodName` - name of the method
- `a, b` - formal parameters
- `int a, int b` - list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax -

#### Syntax

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

The syntax shown above includes -

- `modifier` - It defines the access type of the method and it is optional to use.
- `returnType` - Method may return a value.
- `nameOfMethod` - This is the method name. The method signature consists of the method name and the parameter list.
- `Parameter List` - The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- `method body` - The method body defines what the method does with the statements.

#### Example

Here is the source code of the above defined method called `min()`. This method takes two parameters `num1` and `num2` and returns the maximum between the two -

```
/** the snippet returns the minimum between two numbers */  
public static int minFunction(int n1, int n2) {  
    int min;  
    if (n1 > n2)
```

```
min = n2;
else
    min = n1;
return min;
}
```

### Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when -

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example -  
System.out.println("This is tutorialspoint.com!");

The method returning value can be understood by the following example -  
int result = sum(6, 9);

Following is the example to demonstrate how to define a method and how to call it -  
Examplepublic class ExampleMinNumber {

```
public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
}

/** returns the minimum of two numbers */
public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;
    return min;
}
```

This will produce the following result -

Output

Minimum value = 6

### The void Keyword

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method methodRankPoints. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. methodRankPoints(255.7);. It is a Java statement which ends with a semicolon as shown in

the following example.

### Example

```
public class ExampleVoid {  
    public static void main(String[] args) {  
        methodRankPoints(255.7);  
    }  
    public static void methodRankPoints(double points) {  
        if (points >= 202.5) {  
            System.out.println("Rank:A1");  
        } else if (points >= 122.4) {  
            System.out.println("Rank:A2");  
        } else {  
            System.out.println("Rank:A3");  
        }  
    }  
}
```

This will produce the following result -

### Output

Rank:A1

### Passing Parameters by Value

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

### Example

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```
public class swappingExample {  
    public static void main(String[] args) {  
        int a = 30;  
        int b = 45;  
        System.out.println("Before swapping, a = " + a + " and b = " + b);  
        // Invoke the swap method  
        swapFunction(a, b);  
        System.out.println("\n**Now, Before and After swapping values will be same here**:");  
        System.out.println("After swapping, a = " + a + " and b is " + b);  
    }  
    public static void swapFunction(int a, int b) {  
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);  
  
        // Swap n1 with n2  
        int c = a;  
        a = b;  
        b = c;  
        System.out.println("After swapping(Inside), a = " + a + " b = " + b);  
    }  
}
```

This will produce the following result -

## Output

Before swapping, a = 30 and b = 45  
Before swapping(Inside), a = 30 b = 45  
After swapping(Inside), a = 45 b = 30  
\*\*Now, Before and After swapping values will be same here\*\*:  
After swapping, a = 30 and b is 45

## Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same -

Example

```
public class ExampleOverloading {  
    public static void main(String[] args) {  
        int a = 11;  
        int b = 6;  
        double c = 7.3;  
        double d = 9.4;  
        int result1 = minFunction(a, b);  
  
        // same function name with different parameters  
        double result2 = minFunction(c, d);  
        System.out.println("Minimum Value = " + result1);  
        System.out.println("Minimum Value = " + result2);  
    }  
    // for integer  
    public static int minFunction(int n1, int n2) {  
        int min;  
        if (n1 > n2)  
            min = n2;  
        else  
            min = n1;  
        return min;  
    }  
    // for double  
    public static double minFunction(double n1, double n2) {  
        double min;  
        if (n1 > n2)  
            min = n2;  
        else  
            min = n1;  
        return min;  
    }  
}
```

This will produce the following result -

## Output

Minimum Value = 6

## Minimum Value = 7.3

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

### Using Command-Line Arguments

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to `main()`.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the `String` array passed to `main()`.

### Example

The following program displays all of the command-line arguments that it is called with -

```
public class CommandLine {  
    public static void main(String args[]) {  
        for(int i = 0; i<args.length; i++) {  
            System.out.println("args[" + i + "]: " + args[i]);  
        }  
    }  
}
```

Try executing this program as shown here -

```
$java CommandLine this is a command line 200 -100
```

This will produce the following result -

#### Output

```
args[0]: this  
args[1]: is  
args[2]: a  
args[3]: command  
args[4]: line  
args[5]: 200  
args[6]: -100
```

### The `this` keyword

this is a keyword in Java which is used as a reference to the object of the current class, within an instance method or a constructor. Using `this` you can refer the members of a class such as constructors, variables and methods.

Note - The keyword `this` is used only within instance methods or constructors



In general, the keyword `this` is used to -

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student {  
    int age;  
    Student(int age) {  
        this.age = age;  
    }
```

}

- Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

```
class Student {  
    int age  
    Student() {  
        this(20);  
    }  
}
```

```
Student(int age) {  
    this.age = age;  
}  
}
```

### Example

Here is an example that uses `this` keyword to access the members of a class. Copy and paste the following program in a file with the name, `This_Example.java`.

[Live Demo](#)

```
public class This_Example {  
    // Instance variable num  
    int num = 10;  
  
    This_Example() {  
        System.out.println("This is an example program on keyword this");  
    }  
    This_Example(int num) {  
        // Invoking the default constructor  
        this();  
  
        // Assigning the local variable num to the instance variable num  
        this.num = num;  
    }  
  
    public void greet() {  
        System.out.println("Hi Welcome to Tutorialspoint");  
    }  
  
    public void print() {  
        // Local variable num  
        int num = 20;  
  
        // Printing the local variable  
        System.out.println("value of local variable num is : "+num);  
  
        // Printing the instance variable  
        System.out.println("value of instance variable num is : "+this.num);  
  
        // Invoking the greet method of a class  
        this.greet();  
    }  
  
    public static void main(String[] args) {  
        // Instantiating the class  
        This_Example obj1 = new This_Example();  
  
        // Invoking the print method  
    }  
}
```

```

obj1.print();

// Passing a new value to the num variable through parametrized constructor
This_Example obj2 = new This_Example(30);

// Invoking the print method again
obj2.print();
}
}

```

This will produce the following result -

#### Output

This is an example program on keyword this

value of local variable num is : 20

value of instance variable num is : 10

Hi Welcome to Tutorialspoint

This is an example program on keyword this

value of local variable num is : 20

value of instance variable num is : 30

Hi Welcome to Tutorialspoint

#### Variable Arguments(var-args)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows -

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

#### Example

```

public class VarargsDemo {
public static void main(String args[]) {
    // Call method with variable args
    printMax(34, 3, 3, 2, 56.5);
    printMax(new double[]{1, 2, 3});
}

public static void printMax( double... numbers) {
    if (numbers.length == 0) {
        System.out.println("No argument passed");
        return;
    }
    double result = numbers[0];
    for (int i = 1; i < numbers.length; i++)
        if (numbers[i] > result)
            result = numbers[i];
    System.out.println("The max value is " + result);
}
}

```

This will produce the following result -

#### Output

The max value is 56.5

The max value is 3.0

### The finalize( ) Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called `finalize()`, and it can be used to ensure that an object terminates cleanly.

For example, you might use `finalize()` to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the `finalize()` method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the `finalize()` method, you will specify those actions that must be performed before an object is destroyed.

The `finalize()` method has this general form -

```
protected void finalize() {  
    // finalization code here  
}
```

Here, the keyword `protected` is a specifier that prevents access to `finalize()` by code defined outside its class.

This means that you cannot know when or even if `finalize()` will be executed. For example, if your program ends before garbage collection occurs, `finalize()` will not execute.

# Memory work

30 June 2022 00:55

# Introduction to Arrays

06 January 2022 19:50

## ➤ Why we need Arrays?

- Consider, if we want to store a roll no, we can simply write like-  
`int a = 20;`
- But, imagine --> If we have to store 1000 roll numbers
- Is it possible to write same code 1000 times?  
**Ans is : NO**
- So, For this we use Array data structure.

## ➤ What is an Array?

An array is a group (or collection) of some data types.

## ➤ Syntax of an Array

`datatype [] variable_name = new datatype [size];`

Example --> store 5 roll numbers:

`int [] rollnos = new int[5];`

Or

`int[] rollnos = {5,10,11,12,15};`

 represent the datatype stored in array

 All the type of data in array should be same

Can we proceed without giving size to array?

- No. It is not possible to declare an array without specifying the size
- You can do with ArrayList

## ➤ Internal working of an Array

- `int [] rollnos;` ----> declaration of an array

Here, rollnos are getting defined in stack

- `rollnos = new int[5];` ----> Initialization

Actual memory allocation happens here.

Here, object is being created in heap memory.

## ➤ Dynamic Memory Allocation

Declaration of array  
(at compile time)

`int[]`  
↓  
data type

Initialization  
(at runtime)

`= new int[size]`  
↓  
Creating object in heap memory  
`new` --> a keyword used to create object

## ➤ Internal representation of array





### > Memory allocation in java is continuous or not?

It may be continuous or not. It's totally depends on JVM

Reasons:-

- o Objects are stored in heap memory.
- o According to JLS(Java language specification) it is mentioned that heap objects are not continuous.
- o Dynamic Memory allocation.

### > NOTES

- If we don't assign values in the array, Internally by default it store [0, 0, 0, 0, 0, 0] like this for above arr.
- i.e., for int --> by default value is 0 (zero) for all elements.
- For string ----> by default value is NULL.
- Null --> a literal used for reference.
- Primitive (int, char, float, etc) are stored in Stack.
- Objects are stored in heap memory.

### > Ways to print elements of an array:

1. Using for loop:

```
for(int i=0; i<arr.length; i++){
    System.out.println(arr[i] + " ");
}
```

2. for(int num: arr){

// for-each loop

```
System.out.println(num + " "); // here num represents the element of the array
```

}

3. Using to\_string method of arrays class

```
System.out.println(Arrays.toString(arr));
```

To use this one import  
import java.util.Arrays;

 Internally uses for loop

 BEST WAY

**NOTE:** In JAVA, arrays are mutable(means, object can be changed) and string are immutable.

### > Array passing in Function

```
package com.roshan;
import java.util.Scanner;
public class Demo {
    static int max(int [] array) {
        int max = 0;

        for(int i=0; i<array.length; i++) {
            if(array[i]>max) {
                max = array[i];
            }
        }
    }
}
```

```

    return max;
}

static int min(int [] array) {
    int min = array[0];

    for(int i = 0; i<array.length; i++ ) {
        if(array[i]<min) {
            min = array[i];
        }
    }
    return min;
}

public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the array range");
    int size = sc.nextInt();
    int[] arr = new int[size];
    System.out.println("Enter the elements of the array ::");

    for(int i=0; i<size; i++) {
        arr[i] = sc.nextInt();
    }
    System.out.println("the maximum value in given array is "+max(arr));
    System.out.println("the minimum value in given array is "+min(arr));
}
}

```

Output:

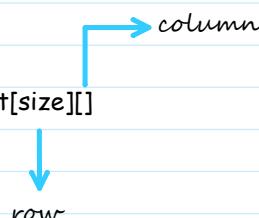
```

Enter the elements of the array ::
1 2 3 4 5
the maximum value in given array is 5
the minimum value in given array is 1

```

### ➤ 2D Array

- Array of arrays
  - Syntax :- int [][] arr = new int[size][]



OR

```

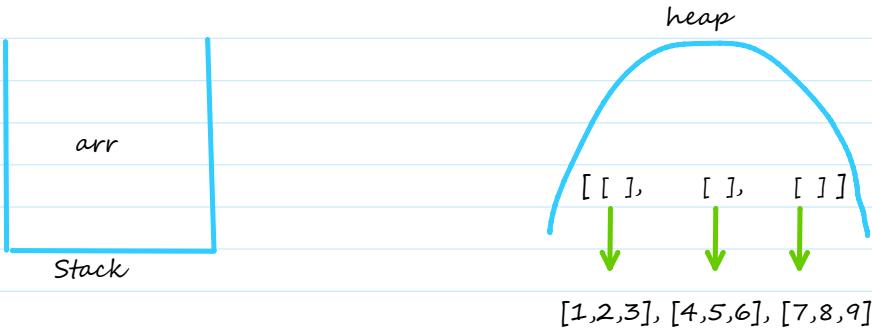
int[][] arr ={
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
}
3*3

```

### NOTE

- It is mandatory to give size of row.
- Not mandatory to give size of column.

## Representation



<code>arr[0] =[1, 2, 3]</code>	<code>arr[0] =[4, 5, 6]</code>
<code>arr[0][0] = 1</code>	<code>arr[0][0] = 4</code>
<code>arr[0][1] = 2</code>	<code>arr[0][1] = 5</code>
<code>arr[0][2] = 3</code>	<code>arr[0][2] = 6</code>

Sample program of 2d array :

```

package com.roshan;

import java.util.Arrays;
import java.util.Scanner;

public class Demo2 {
    public static void main(String[] args) {
        Scanner in= new Scanner(System.in);
        int [][] arr ={{1,2,3},
                      {4,5},
                      {6,7,8}};

        //arr.length = it will give no of rows
        for (int row = 0; row < arr.length; row++) {
            for (int col = 0; col < arr[row].length ; col++) {
                System.out.print(arr[row][col]+" ");
            }
            System.out.println();
        }

        for (int row = 0; row < arr.length; row++) {
            System.out.println(Arrays.toString(arr[row]));
        }
    }
}

```

1 2 3  
4 5  
6 7 8

[1, 2, 3]  
[4, 5]  
[6, 7, 8]

## ArrayList

16 January 2022 14:26

### > Why we need ArrayList?

We have to give size while initialization. But, what if we don't know the size of array?  
Then, we use array list.

### > ArrayList

- It is part of collection framework.
- It is present in java.util.package.
- It provides dynamic arrays
- It is slower than standard arrays

Syntax

```
ArrayList<data type> list = new ArrayList<>();
```



[like integer (not int)]

i.e., add wrappers

### > Internal Working of ArrayList

- Size is fixed internally.
- If ArrayList gets filled by some amount then --->
  - It will make a new ArrayList of say, double the size of order arraylist.
  - Old elements are copied in new ArrayList.
  - Old ones are deleted.

```
//sample program --- array list
```

```
package com.array2;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

/*
import java.util.*;
it will add all the classes which are there in java
drawback - it will be adding class which we don't need in ongoing file
the more class in the file the more size of the file will be that's why we should avoid writing this
*/
```

```
public class ArrayLists {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        ArrayList<String> list2 = new ArrayList<>();

        //add elements
        list.add(0);
        list.add(211);
```

```
list.add(213);
list.add(33);

System.out.println(list); // [0, 211, 213, 33]

//get elements
int element=list.get(0);
System.out.println(element); //0

//add element in between
list.add(1,1);
System.out.println(list); // [0, 1, 211, 213, 33]

//set element
list.set(0,5);
System.out.println(list); // [5, 1, 211, 213, 33]

//delete element
list.remove(0);
System.out.println(list); // [1, 211, 213, 33]
```

There are many methods but main main methods which you will mostly use in java array list has shown in this program

```
//size
System.out.println(list.size()); //4

//loops
for (int i = 0; i < list.size(); i++) {
    System.out.print(list.get(i)+ " ");
} // 1 211 213 33

System.out.println();

//sorting
Collections.sort(list); //this class is applicable for arraylist only for array it has different
System.out.println(list); // [1, 33, 211, 213]
```

```
}
```

# All methods of ArrayList

16 January 2022 15:53

## ➤ Methods in Java ArrayList

Method	Description
<code>add(int index, Object element)</code>	This method is used to insert a specific element at a specific position index in a list.
<code>add(Object o)</code>	This method is used to append a specific element to the end of a list.
<code>addAll(Collection C)</code>	This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection's iterator.
<code>addAll(int index, Collection C)</code>	Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list.
<code>clear()</code>	This method is used to remove all the elements from any list.
<code>? clone()</code>	This method is used to return a shallow copy of an ArrayList.
<code>contains?(Object o)</code>	Returns true if this list contains the specified element.
<code>ensureCapacity?(int minCapacity)</code>	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
<code>forEach?(Consumer&lt;? super E&gt; action)</code>	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
<code>get?(int index)</code>	Returns the element at the specified position in this list.
<code>indexOf(Object O)</code>	The index the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.
<code>isEmpty?()</code>	Returns true if this list contains no elements.
<code>lastIndexOf(Object O)</code>	The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list.
<code>listIterator?()</code>	Returns a list iterator over the elements in this list (in proper sequence).
<code>listIterator?(int index)</code>	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
<code>remove?(int index)</code>	Removes the element at the specified position in this list.
<code>remove?(Object o)</code>	Removes the first occurrence of the specified element from this list, if it is present.
<code>removeAll?(Collection c)</code>	Removes from this list all of its elements that are contained in the specified collection.
<code>removeIf?(Predicate filter)</code>	Removes all of the elements of this collection that satisfy the given predicate.
<code>removeRange?(int fromIndex, int toIndex)</code>	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
<code>retainAll?(Collection nc)</code>	Retains only the elements in this list that are contained in the specified collection.
<code>set?(int index, E element)</code>	Replaces the element at the specified position in this list with the specified element.
<code>size?()</code>	Returns the number of elements in this list.

<code>spliterator()</code>	Creates a late-binding and fail-fast Spliterator over the elements in this list.
<code>subList(int fromIndex, int toIndex)</code>	Returns a view of the portion of this list between the specified <code>fromIndex</code> , inclusive, and <code>toIndex</code> , exclusive.
<code>toArray()</code>	This method is used to return an array containing all of the elements in the list in the correct order.
<code>toArray(Object[] O)</code>	It is also used to return an array containing all of the elements in this list in the correct order same as the previous method.
<code>trimToSize()</code>	This method is used to trim the capacity of the instance of the <code>ArrayList</code> to the list's current size.

### ➤ add()

```

import java.io.*;
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String[] args) {
        // create an empty array list with an initial capacity
        ArrayList<Integer> arrlist = new ArrayList<Integer>(5);
        Number = 15
        Number = 20
        Number = 25

        // use add() method to add elements in the list
        arrlist.add(15);
        arrlist.add(20);
        arrlist.add(25);

        // prints all the elements available in list
        for (Integer number : arrlist) {
            System.out.println("Number = " + number);
        }
    }
}

```

### ➤ add(int index, Object element) :

```

// Java code to illustrate
// void add(int index, Object element)
import java.io.*;
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String[] args) {
        // create an empty array list with an initial capacity
        ArrayList<Integer> arrlist = new ArrayList<Integer>(5);
        Number = 10
        Number = 22
        Number = 30
        Number = 35
        Number = 40

        // use add() method to add elements in the list
        arrlist.add(10);
        arrlist.add(22);
        arrlist.add(30);
    }
}

```

```

arrlist.add(40);

// adding element 35 at fourth position
arrlist.add(3, 35);

// let us print all the elements available in list
for (Integer number : arrlist) {
    System.out.println("Number = " + number);
}
}
}

```

➤ addAll()

```

// Java program to illustrate
// boolean addAll(Collection c)
import java.io.*;
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String args[])
    {

        // create an empty array list1 with initial
        // capacity as 5
        ArrayList<Integer> arrlist1 =
            new ArrayList<Integer>(5);

        // use add() method to add elements in the list
        arrlist1.add(12);
        arrlist1.add(20);
        arrlist1.add(45);

        // prints all the elements available in list1
        System.out.println("Printing list1:");
        for (Integer number : arrlist1)
            System.out.println("Number = " + number);

        // create an empty array list2 with an initial
        // capacity
        ArrayList<Integer> arrlist2 =
            new ArrayList<Integer>(5);

        // use add() method to add elements in list2
        arrlist2.add(25);
        arrlist2.add(30);
        arrlist2.add(31);
        arrlist2.add(35);

        // let us print all the elements available in
        // list2
        System.out.println("Printing list2:");
        for (Integer number : arrlist2)
            System.out.println("Number = " + number);
    }
}

```

Output:  
 Printing list1:  
 Number = 12  
 Number = 20  
 Number = 45  
 Printing list2:  
 Number = 25  
 Number = 30  
 Number = 31  
 Number = 35  
 Printing all the elements  
 Number = 12  
 Number = 20  
 Number = 45  
 Number = 25  
 Number = 30  
 Number = 31  
 Number = 35

```

// inserting all elements, list2 will get printed
// after list1
arrlist1.addAll(arrlist2);

System.out.println("Printing all the elements");
// let us print all the elements available in
// list1
for (Integer number : arrlist1)
    System.out.println("Number = " + number);
}
}

```

➤ [addAll\(int index, Collection c\)](#)

```

// Java program to illustrate
// boolean addAll(int index, Collection c)
import java.io.*;
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String args[])
    {

        // create an empty array list1 with initial
        // capacity 5
        ArrayList<Integer> arrlist =
            new ArrayList<Integer>(5);

        // using add() method to add elements in the
        // list
        arrlist.add(12);
        arrlist.add(20);
        arrlist.add(45);

        // prints all the elements available in list1
        System.out.println("Printing list1:");
        for (Integer number : arrlist)
            System.out.println("Number = " + number);

        // create an empty array list2 with an initial
        // capacity
        ArrayList<Integer> arrlist2 =
            new ArrayList<Integer>(5);

        // use add() method to add elements in list2
        arrlist2.add(25);
        arrlist2.add(30);
        arrlist2.add(31);
        arrlist2.add(35);

        // prints all the elements available in list2
        System.out.println("Printing list2:");
        for (Integer number : arrlist2)
            System.out.println("Number = " + number);
    }
}

```

Output:  
 Printing list1:  
 Number = 12  
 Number = 20  
 Number = 45  
 Printing list2:  
 Number = 25  
 Number = 30  
 Number = 31

```

for (Integer number : arrlist2)
    System.out.println("Number = " + number);

// inserting all elements of list2 at third
// position
arrlist.addAll(2, arrlist2);

System.out.println("Printing all the elements");

// prints all the elements available in list1
for (Integer number : arrlist)
    System.out.println("Number = " + number);

}

}


```

Number = 30  
Number = 31  
Number = 35  
Printing all the elements  
Number = 12  
Number = 20  
Number = 25  
Number = 30  
Number = 31  
Number = 35  
Number = 45

➤ [clear\(\)](#)

```

// Java Program to Illustrate Working of clear() Method
// of ArrayList class

// Importing required classes
import java.util.ArrayList;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(4);

        // Adding elements to above ArrayList
        // using add() method
        arr.add(1);
        arr.add(2);
        arr.add(3);
        arr.add(4);

        // Printing the elements inside current ArrayList
        System.out.println("The list initially: " + arr);

        // Clearing off elements
        // using clear() method
        arr.clear();

        // Displaying ArrayList elements
        // after using clear() method
        System.out.println(
            "The list after using clear() method: " + arr);
    }
}


```

➤ [Clone\(\)](#)

```

// Java program to demonstrate that assignment operator
// only creates a new reference to same object
import java.io.*;

// A test class whose objects are cloned
class Test {
    int x, y;
    Test()
    {
        x = 10;
        y = 20;
    }
}

// Driver Class
class Main {
    public static void main(String[] args)
    {
        Test ob1 = new Test();

        System.out.println(ob1.x + " " + ob1.y);

        // Creating a new reference variable ob2
        // pointing to same address as ob1
        Test ob2 = ob1;

        // Any change made in ob2 will
        // be reflected in ob1
        ob2.x = 100;

        System.out.println(ob1.x + " " + ob1.y);
        System.out.println(ob2.x + " " + ob2.y);
    }
}

```

The list initially: [1, 2, 3, 4]  
The list after using clear()  
method: []

From <<https://www.geeksforgeeks.org/arraylist-clear-method-in-java-with-examples/>>

10 20  
100 20  
100 20

From  
<<https://www.geeksforgeeks.org/clone-method-in-java-2/>>

## All methods of ArrayList 2

18 January 2022 01:25

### ➤ ArrayList.contains() in Java

```
// Java code to demonstrate the working of
// contains() method in ArrayList

// for ArrayList functions
import java.util.ArrayList;

class GFG {
    public static void main(String[] args)
    {

        // creating an Empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(4);

        // using add() to initialize values
        // [1, 2, 3, 4]
        arr.add(1);
        arr.add(2);
        arr.add(3);
        arr.add(4);

        // use contains() to check if the element
        // 2 exists or not
        boolean ans = arr.contains(2);

        if (ans)
            System.out.println("The list contains 2");
        else
            System.out.println("The list does not contain 5");

        // use contains() to check if the element
        // 5 exists or not
        ans = arr.contains(5);

        if (ans)
            System.out.println("The list contains 5");
        else
            System.out.println("The list does not contain 5");
    }
}
```

The list contains 2

The list does not contain 5

### ➤ ArrayList.ensureCapacity() method in Java

```
// Java program to demonstrate
// ensureCapacity() method for Integer value

import java.util.*;

public class GFG1 {
    public static void main(String[] argv)
        throws Exception
    {

        try {
```

```

// Creating object of ArrayList<Integer>
ArrayList<Integer>
    arrlist = new ArrayList<Integer>();

// adding element to arrlist
arrlist.add(10);
arrlist.add(20);
arrlist.add(30);
arrlist.add(40);

// Print the ArrayList
System.out.println("ArrayList: "
    + arrlist);

// ensure that the ArrayList
// can hold upto 5000 elements
// using ensureCapacity() method
arrlist.ensureCapacity(5000);

// Print
System.out.println("ArrayList can now"
    + " surely store upto"
    + " 5000 elements.");
}

catch (NullPointerException e) {
    System.out.println("Exception thrown : " + e);
}
}

}

ArrayList: [10, 20, 30, 40]
ArrayList can now surely store upto
5000 elements.

```

### ArrayList forEach() method in Java

From <<https://www.geeksforgeeks.org/arraylist-foreach-method-in-java/>>

```

// Java Program Demonstrate forEach()
// method of ArrayList

import java.util.*;
public class GFG {

    public static void main(String[] args)
    {
        // create an ArrayList which going to
        // contains a list of Numbers
        ArrayList<Integer> Numbers = new ArrayList<Integer>();
        23
        32
        45
        63

        // Add Number to list
        Numbers.add(23);
        Numbers.add(32);
        Numbers.add(45);
        Numbers.add(63);

        // forEach method of ArrayList and
        // print numbers
        Numbers.forEach((n) -> System.out.println(n));
    }
}

```

### ArrayList get(index) Method in Java

From <<https://www.geeksforgeeks.org/arraylist-get-method-java-examples/>>

```
// Java Program to Demonstrate the working of
// get() method in ArrayList

// Importing ArrayList class
import java.util.ArrayList;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an Empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(4);

        // Using add() to initialize values
        // [10, 20, 30, 40]
        arr.add(10);
        arr.add(20);
        arr.add(30);
        arr.add(40);                                         List: [10, 20, 30, 40]
                                                               the element at index 2 is 30

        // Printing elements of list
        System.out.println("List: " + arr);

        // Getting element at index 2
        int element = arr.get(2);

        // Displaying element at specified index
        // on console inside list
        System.out.println("the element at index 2 is "
                           + element);
    }
}
```

### Java.util.ArrayList.indexOf() in Java

From <<https://www.geeksforgeeks.org/java-util-arraylist-indexof-java/>>

```
// Java code to demonstrate the working of
// indexOf in ArrayList

// for ArrayList functions
import java.util.ArrayList;

public class IndexOfEx {
    public static void main(String[] args) {

        // creating an Empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(5);

        // using add() to initialize values
        arr.add(1);
        arr.add(2);
        arr.add(3);
        arr.add(4);
```

```

// printing initial value
System.out.print("The initial values in ArrayList are : ");
for (Integer value : arr) {
    System.out.print(value);
    System.out.print(" ");
}

// using indexOf() to find index of 3
int pos = arr.indexOf(3);           The initial values in ArrayList
                                    are : 1 2 3 4
                                    The element 3 is at index : 2

// prints 2
System.out.println("\nThe element 3 is at index : " + pos);
}

}

```

### ArrayList isEmpty() in Java

From <<https://www.geeksforgeeks.org/arraylist-isempty-java-example/>>

```

// Java code to demonstrate the working of
// isEmpty() method in ArrayList

// for ArrayList functions
import java.util.ArrayList;

public class GFG {
    public static void main(String[] args)
    {

        // creating an Empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(10);

        // check if the list is empty or not using function
        boolean ans = arr.isEmpty();
        if (ans == true)
            System.out.println("The ArrayList is empty");
        else
            System.out.println("The ArrayList is not empty");

        // addition of a element to the ArrayList
        arr.add(1);                           The ArrayList is empty
                                            The ArrayList is not empty

        // check if the list is empty or not
        ans = arr.isEmpty();
        if (ans == true)
            System.out.println("The ArrayList is empty");
        else
            System.out.println("The ArrayList is not empty");
    }
}

```

### ArrayList lastIndexOf() in Java

From <<https://www.geeksforgeeks.org/arraylist-lastindexof-java-example/>>

```

// Java code to demonstrate the working of
// lastIndexOf() method in ArrayList

// for ArrayList functions

```

```

import java.util.ArrayList;

public class GFG {
    public static void main(String[] args)
    {

        // creating an Empty Integer ArrayList
        ArrayList<Integer> arr = new ArrayList<Integer>(7);

        // using add() to initialize values
        arr.add(10);
        arr.add(20);
        arr.add(30);
        arr.add(40);
        arr.add(30);
        arr.add(30);
        arr.add(40);

        System.out.println("The list initially " + arr);

        // last index of 30

        int element = arr.lastIndexOf(30);
        if (element != -1)
            System.out.println("the lastIndexof of " +
                               " 30 is " + element);
        else
            System.out.println("30 is not present in" +
                               " the list");

        // last index of 100
        element = arr.lastIndexOf(100);
        if (element != -1)
            System.out.println("the lastIndexof of 100" +
                               " is " + element);
        else
            System.out.println("100 is not present in" +
                               " the list");
    }
}

```

The list initially [10, 20, 30, 40, 30,  
30, 40]  
the lastIndexof of 30 is 5  
100 is not present in the list

### ArrayList listIterator() method in Java

From <<https://www.geeksforgeeks.org/arraylist-listiterator-method-in-java-with-examples/>>

```

// Java program to demonstrate
// listIterator() method
// for String value

import java.util.*;

public class GFG1 {
    public static void main(String[] argv) throws Exception
    {
        try {

            // Creating object of ArrayList<Integer>
            ArrayList<String>
                arrlist = new ArrayList<String>();

```

```

// adding element to arrlist
arrlist.add("A");
arrlist.add("B");
arrlist.add("C");
arrlist.add("D");

// print arrlist
System.out.println("ArrayList: "
+ arrlist);

// Creating object of ListIterator<String>
// using listIterator() method
ListIterator<String>
    iterator = arrlist.listIterator();

    // Printing the iterated value
    System.out.println("\nUsing ListIterator:\n");
    while (iterator.hasNext()){
        System.out.println("Value is : "
        + iterator.next());
    }
}

catch (NullPointerException e){
    System.out.println("Exception thrown : " + e);
}
}
}

```

ArrayList: [A, B, C, D]  
Using ListIterator:  
Value is : A  
Value is : B  
Value is : C  
Value is : D

### listIterator(int index)

From <<https://www.geeksforgeeks.org/arraylist-listiterator-method-in-java-with-examples/>>

```

// Java program to demonstrate
// listIterator() method
// for String value

import java.util.*;

public class GFG1 {
    public static void main(String[] argv) throws Exception
    {
        try {

            // Creating object of ArrayList<Integer>
            ArrayList<String> arrlist = new ArrayList<String>();

            // adding element to arrlist
            arrlist.add("A");
            arrlist.add("B");
            arrlist.add("C");
            arrlist.add("D");

            // print arrlist
            System.out.println("ArrayList: "
            + arrlist);

            // getting iterated value starting from index 2

```

```

// using listIterator() method
ListIterator<String>
    iterator = arrlist.listIterator(2);

// Printing the iterated value
System.out.println("\nUsing ListIterator"
    + " from Index 2:\n");
while (iterator.hasNext()) {
    System.out.println("Value is : "
        + iterator.next());
}
}

catch (IndexOutOfBoundsException e) {
    System.out.println("Exception thrown : " + e);
}
}
}

```

ArrayList: [A, B, C, D]  
Using ListIterator from Index 2:  
Value is : C  
Value is : D

### **ArrayList and LinkedList remove() methods in Java**

From <<https://www.geeksforgeeks.org/arraylist-linkedlist-remove-methods-java-examples/>>

```

// A Java program to demonstrate working of list remove
// when Object to be removed is passed.
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        // Demonstrating remove on ArrayList
        List<String> myAlist = new ArrayList<String>();
        myAlist.add("Geeks");
        myAlist.add("Practice");
        myAlist.add("Quiz");
        System.out.println("Original ArrayList : " + myAlist);
        myAlist.remove("Quiz");
        System.out.println("Modified ArrayList : " + myAlist);

        // Demonstrating remove on LinkedList
        List<String> myList = new LinkedList<String>();
        myList.add("Geeks");
        myList.add("Practice");
        myList.add("Quiz");
        System.out.println("Original LinkedList : " + myList);
        myList.remove("Quiz");
        System.out.println("Modified LinkedList : " + myList);
    }
}

```

Original ArrayList : [Geeks, Practice, Quiz]  
Modified ArrayList : [Geeks, Practice]  
Original LinkedList : [Geeks, Practice, Quiz]  
Modified LinkedList : [Geeks, Practice]

### **ArrayList removeIf() method in Java**

From <<https://www.geeksforgeeks.org/arraylist-removeif-method-in-java/>>

```

// Java Program Demonstrate removeIf()
// method of ArrayList

import java.util.*;
public class GFG {

```

```
public static void main(String[] args)
{
    // create an ArrayList which going to
    // contains a list of Numbers
    ArrayList<Integer> Numbers = new ArrayList<Integer>();

    // Add Number to list
    Numbers.add(23);
    23
    32
```

From <<https://www.geeksforgeeks.org/arraylist-removeif-method-in-java/>>

```
Numbers.add(32);
Numbers.add(45);
Numbers.add(63);

// apply removeIf() method
// we are going to remove numbers divisible by 3
Numbers.removeIf(n -> (n % 3 == 0));

// print list
for (int i : Numbers) {
    System.out.println(i);
}
}
```

### ArrayList removeRange() in Java

From <<https://www.geeksforgeeks.org/arraylist-removerange-java-examples/>>

```
// Java program to demonstrate the
// working of removeRange() method
import java.util.*;

// extending the class to arraylist since removeRange()
// is a protected method
public class GFG extends ArrayList<Integer> {

    public static void main(String[] args)
    {

        // create an empty array list
        GFG arr = new GFG();

        // use add() method to add values in the list
        arr.add(1);
        arr.add(2);
        arr.add(3);
        arr.add(12);
        arr.add(9);
        arr.add(13);

        // prints the list before removing
        System.out.println("The list before using removeRange:" + arr);
```

```

        // removing range of 1st 2 elements
        arr.removeRange(0, 2);
        System.out.println("The list after using removeRange:" + arr);
    }
}

```

### ArrayList retainAll() method in Java

From <<https://www.geeksforgeeks.org/arraylist-retainall-method-in-java/>>

```

// Java code to illustrate retainAll() method
import java.util.ArrayList;
public class GFG {
    public static void main(String[] args)
    {
        // Creating an empty array list
        ArrayList<String> bags = new ArrayList<String>();

        // Add values in the bags list.
        bags.add("pen");
        bags.add("pencil");
        bags.add("paper");

        // Creating another array list
        ArrayList<String> boxes = new ArrayList<String>();

        // Add values in the boxes list.
        boxes.add("pen");
        boxes.add("paper");
        boxes.add("books");
        boxes.add("rubber");

        // Before Applying method print both lists
        System.out.println("Bags Contains :" + bags);
        System.out.println("Boxes Contains :" + boxes);

        // Apply retainAll() method to boxes passing bags as parameter
        boxes.retainAll(bags);

        // Displaying both the lists after operation
        System.out.println("\nAfter Applying retainAll()"+
        " method to Boxes\n");
        System.out.println("Bags Contains :" + bags);
        System.out.println("Boxes Contains :" + boxes);
    }
}

```

### ArrayList set() method in Java

From <<https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/>>

```

// Java program to demonstrate
// set() method
// for Integer value

import java.util.*;

public class GFG1 {
    public static void main(String[] argv) throws Exception
    {

```

```

try {

    // Creating object of ArrayList<Integer>
    ArrayList<Integer>
        arrlist = new ArrayList<Integer>();

    // Populating arrlist1
    arrlist.add(1);
    arrlist.add(2);
    arrlist.add(3);
    arrlist.add(4);
    arrlist.add(5);

    // print arrlist
    System.out.println("Before operation: "
        + arrlist);

    // Replacing element at the index 3 with 30
    // using method set()
    int i = arrlist.set(3, 30);

    // Print the modified arrlist
    System.out.println("After operation: "
        + arrlist);

    // Print the Replaced element
    System.out.println("Replaced element: "
        + i);
}

catch (IndexOutOfBoundsException e) {
    System.out.println("Exception thrown: "
        + e);
}
}
}

```

### ArrayList size() method in Java

From <<https://www.geeksforgeeks.org/arraylist-size-method-in-java-with-examples/>>

```

// Java program to demonstrate
// size() method
// for Integer value

import java.util.*;

public class GFG1 {
    public static void main(String[] argv)
        throws Exception
    {
        try {

            // Creating object of ArrayList<Integer>
            ArrayList<Integer>
                arrlist = new ArrayList<Integer>();

            // Populating arrlist1
            arrlist.add(1);
            arrlist.add(2);

```

```

        arrlist.add(3);
        arrlist.add(4);
        arrlist.add(5);

        // print arrlist
        System.out.println("Before operation: "
                           + arrlist);

        // getting total size of arrlist
        // using size() method
        int size = arrlist.size();

        // print the size of arrlist
        System.out.println("Size of list = "
                           + size);
    }

    catch (IndexOutOfBoundsException e) {

        System.out.println("Exception thrown: "
                           + e);
    }
}
}

```

### **ArrayList spliterator() method in Java**

From <<https://www.geeksforgeeks.org/arraylist-spliterator-method-in-java/>>

```

// Java Program Demonstrate spliterator()
// method of ArrayList

import java.util.*;

public class GFG {
    public static void main(String[] args)
    {

        // create an ArrayList which contains
        // emails for a group of people
        ArrayList<String> list = new ArrayList<String>();

        // Add emails to list
        list.add("abc@geeksforgeeks.org");
        list.add("user@geeksforgeeks.org");
        list.add("pqr@geeksforgeeks.org");
        list.add("random@geeksforgeeks.org");
        list.add("randomuser@geeksforgeeks.org");

        // create Spliterator of ArrayList
        // using spliterator() method
        Spliterator<String> emails = listspliterator();

        // print result from Spliterator
        System.out.println("list of Emails:");

        // forEachRemaining method of Spliterator
        emails.forEachRemaining((n) -> System.out.println(n));
    }
}

```

## ArrayList subList() method in Java

From <<https://www.geeksforgeeks.org/arraylist-sublist-method-in-java-with-examples/>>

```
// Java program to demonstrate
// subList() method
// for String value

import java.util.*;

public class GFG1 {
    public static void main(String[] argv)
        throws Exception
    {

        try {

            // Creating object of ArrayList<Integer>
            ArrayList<String>
                arrlist = new ArrayList<String>();

            // Populating arrlist1
            arrlist.add("A");
            arrlist.add("B");
            arrlist.add("C");
            arrlist.add("D");
            arrlist.add("E");

            // print arrlist
            System.out.println("Original arrlist: "
                + arrlist);

            // getting the subList
            // using subList() method
            List<String> arrlist2 = arrlist.subList(2, 4);

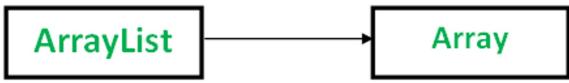
            // print the subList
            System.out.println("Sublist of arrlist: "
                + arrlist2);
        }

        catch (IndexOutOfBoundsException e) {
            System.out.println("Exception thrown : " + e);
        }

        catch (IllegalArgumentException e) {
            System.out.println("Exception thrown : " + e);
        }
    }
}
```

## ArrayList to Array Conversion in Java : toArray() Methods

From <<https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods/>>



#### Methods:

1. `Object[] toArray()`
2. `T[] toArray(T[] a)`
3. `get() method`

```

// Java program to demonstrate working of
// Object[] toArray()
import java.io.*;
import java.util.List;
import java.util.ArrayList;

class GFG {
    public static void main(String[] args)
    {
        List<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);

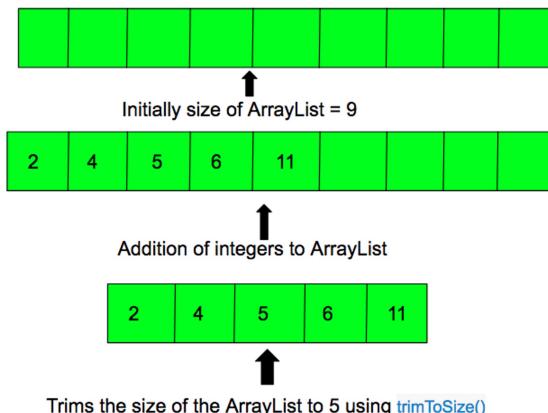
        Object[] objects = al.toArray();

        // Printing array of objects
        for (Object obj : objects)
            System.out.print(obj + " ");
    }
}

```

### ArrayList trimToSize() in Java

From <<https://www.geeksforgeeks.org/arraylist-trimtosize-java-example/>>



```

// Java code to demonstrate the working of
// trimToSize() method in ArrayList

// for ArrayList functions
import java.util.ArrayList;

public class GFG {
    public static void main(String[] args)
    {

        // creating an Empty Integer ArrayList

```

```
ArrayList<Integer> arr = new ArrayList<Integer>(9);

// using add(), add 5 values
arr.add(2);
arr.add(4);
arr.add(5);
arr.add(6);
arr.add(11);

// trims the size to the number of elements
arr.trimToSize();

System.out.println("The List elements are:");

// prints all the elements
for (Integer number : arr) {
    System.out.println("Number = " + number);
}

}
```

Original arrlist: [A, B, C, D, E]  
Sublist of arrlist: [C, D]

From <<https://www.geeksforgeeks.org/arraylist-sublist-method-in-java-with-examples/>>

10 20 30 40

From <<https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods/>>

The List elements are:  
Number = 2  
Number = 4  
Number = 5  
Number = 6  
Number = 11

From <<https://www.geeksforgeeks.org/arraylist-trimtosome-size-javascript-example/>>

# ArrayList Vs Linked list

18 January 2022 02:28

ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contiguous.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the List interface.

## Java list

23 June 2022 00:48

List in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the `java.util` package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

Method	Description
<code>void add(int index, E element)</code>	It is used to insert the specified element at the specified position in a list.
<code>boolean add(E e)</code>	It is used to append the specified element at the end of a list.
<code>boolean addAll(Collection&lt;? extends E&gt; c)</code>	It is used to append all of the elements in the specified collection to the end of a list.
<code>boolean addAll(int index, Collection&lt;? extends E&gt; c)</code>	It is used to append all the elements in the specified collection, starting at the specified position of the list.
<code>void clear()</code>	It is used to remove all of the elements from this list.
<code>boolean equals(Object o)</code>	It is used to compare the specified object with the elements of a list.
<code>int hashCode()</code>	It is used to return the hash code value for a list.
<code>E get(int index)</code>	It is used to fetch the element from the particular position of the list.
<code>boolean isEmpty()</code>	It returns true if the list is empty, otherwise false.
<code>int lastIndexOf(Object o)</code>	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
<code>Object[] toArray()</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>&lt;T&gt; T[] toArray(T[] a)</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>boolean contains(Object o)</code>	It returns true if the list contains the specified element
<code>boolean containsAll(Collection&lt;?&gt; c)</code>	It returns true if the list contains all the specified element
<code>int indexOf(Object o)</code>	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
<code>E remove(int index)</code>	It is used to remove the element present at the specified position in the list.
<code>boolean remove(Object o)</code>	It is used to remove the first occurrence of the specified element.
<code>boolean removeAll(Collection&lt;?&gt; c)</code>	It is used to remove all the elements from the list.
<code>void replaceAll(UnaryOperator&lt;E</code>	It is used to replace all the elements from the list with the specified element.

> operator)	
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

### Java List vs ArrayList

List is an interface whereas ArrayList is the implementation class of List

#### How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

```
//Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();
```

```
//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();
```

```
//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();
```

```
//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

In short, you can create the List of any type. The ArrayList<T> and LinkedList<T> classes are used to specify the type. Here, T denotes the type.

#### Java List Example

Let's see a simple example of List where we are using the ArrayList class as the implementation.

```
import java.util.*;
public class ListExample1{
    public static void main(String args[]){
        //Creating a List
        List<String> list=new ArrayList<String>();
        //Adding elements in the List
        list.add("Mango");
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
        //Iterating the List element using for-each loop
        for(String fruit:list)
            System.out.println(fruit);
```

```
}
```

### How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

```
import java.util.*;
public class ArrayToListExample{
    public static void main(String args[]){
        //Creating Array
        String[] array={"Java","Python","PHP","C++"};
        System.out.println("Printing Array: "+Arrays.toString(array));
        //Converting Array to List
        List<String> list=new ArrayList<String>();
        for(String lang:array){
            list.add(lang);
        }
        System.out.println("Printing List: "+list);
    }
}
```

#### Output:

```
Printing Array: [Java, Python, PHP, C++]
Printing List: [Java, Python, PHP, C++]
```

### How to convert List to Array

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

```
import java.util.*;
public class ListToArrayExample{
    public static void main(String args[]){
        List<String> fruitList = new ArrayList<>();
        fruitList.add("Mango");
        fruitList.add("Banana");
        fruitList.add("Apple");
        fruitList.add("Strawberry");
        //Converting ArrayList to Array
        String[] array = fruitList.toArray(new String[fruitList.size()]);
        System.out.println("Printing Array: "+Arrays.toString(array));
        System.out.println("Printing List: "+fruitList);
    }
}
```

#### Output:

```
Printing Array: [Mango, Banana, Apple, Strawberry]
Printing List: [Mango, Banana, Apple, Strawberry]
```

### Get and Set Element in List

The get() method returns the element at the given index, whereas the set() method changes or replaces the element.

```

import java.util.*;
public class ListExample2{
    public static void main(String args[]){
        //Creating a List
        List<String> list=new ArrayList<String>();
        //Adding elements in the List
        list.add("Mango");
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
        //accessing the element
        System.out.println("Returning element: "+list.get(1)); //it will return the 2nd element, because index starts from 0
        //changing the element
        list.set(1,"Dates");
        //Iterating the List element using for-each loop
        for(String fruit:list)
            System.out.println(fruit);
    }
}

```

#### Output:

Returning element: Apple  
 Mango  
 Dates  
 Banana  
 Grapes

#### How to Sort List

There are various ways to sort the List, here we are going to use `Collections.sort()` method to sort the list element. The `java.util` package provides a utility class `Collections` which has the static method `sort()`. Using the `Collections.sort()` method, we can easily sort any List.

```

import java.util.*;
class SortArrayList{
    public static void main(String args[]){
        //Creating a list of fruits
        List<String> list1=new ArrayList<String>();
        list1.add("Mango");
        list1.add("Apple");
        list1.add("Banana");
        list1.add("Grapes");
        //Sorting the list
        Collections.sort(list1);
        //Traversing list through the for-each loop
        for(String fruit:list1)
            System.out.println(fruit);

        System.out.println("Sorting numbers...");
        //Creating a list of numbers
        List<Integer> list2=new ArrayList<Integer>();
        list2.add(21);
        list2.add(11);
        list2.add(51);
        list2.add(1);
        //Sorting the list
    }
}

```

```

Collections.sort(list2);
//Traversing list through the for-each loop
for(Integer number:list2)
    System.out.println(number);
}
}

```

**Output:**

Apple  
Banana  
Grapes  
Mango  
Sorting numbers...

1  
11  
21  
51

### Java ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.  
ListIterator Interface declaration

public interface ListIterator<E> extends Iterator<E>

Methods of Java ListIterator Interface:

Method	Description
void add(E e)	This method inserts the specified element into the list.
boolean hasNext()	This method returns true if the list iterator has more elements while traversing the list in the forward direction.
E next()	This method returns the next element in the list and advances the cursor position.
int nextIndex()	This method returns the index of the element that would be returned by a subsequent call to next()
boolean hasPrevious()	This method returns true if this list iterator has more elements while traversing the list in the reverse direction.
E previous()	This method returns the previous element in the list and moves the cursor position backward.
E previousIndex()	This method returns the index of the element that would be returned by a subsequent call to previous().
void remove()	This method removes the last element from the list that was returned by next() or previous() methods
void set(E e)	This method replaces the last element returned by next() or previous() methods with the specified element.

Example of ListIterator Interface

```

import java.util.*;
public class ListIteratorExample1{

```

```

public static void main(String args[]){
    List<String> al=new ArrayList<String>();
    al.add("Amit");
    al.add("Vijay");
    al.add("Kumar");
    al.add(1,"Sachin");
    ListIterator<String> itr=al.listIterator();
    System.out.println("Traversing elements in forward direction");
    while(itr.hasNext()){
        System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());
    }
    System.out.println("Traversing elements in backward direction");
    while(itr.hasPrevious()){
        System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());
    }
}
}

```

*Output:*

Traversing elements in forward direction

```

index:0 value:Amit
index:1 value:Sachin
index:2 value:Vijay
index:3 value:Kumar

```

Traversing elements in backward direction

```

index:3 value:Kumar
index:2 value:Vijay
index:1 value:Sachin
index:0 value:Amit

```

### *Example of List: Book*

Let's see an example of List where we are adding the Books.

```

import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int quantity;
    public Book(int id, String name, String author, String publisher, int quantity) {
        this.id = id;
        this.name = name;
        this.author = author;
        this.publisher = publisher;
        this.quantity = quantity;
    }
}
public class ListExample5 {
    public static void main(String[] args) {
        //Creating list of Books
        List<Book> list=new ArrayList<Book>();
        //Creating Books
        Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
        Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
        Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
        //Adding Books to list
    }
}

```

```
list.add(b1);
list.add(b2);
list.add(b3);
//Traversing list
for(Book b:list){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
}
}
```

**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# SEARCHING

27 November 2021 13:28

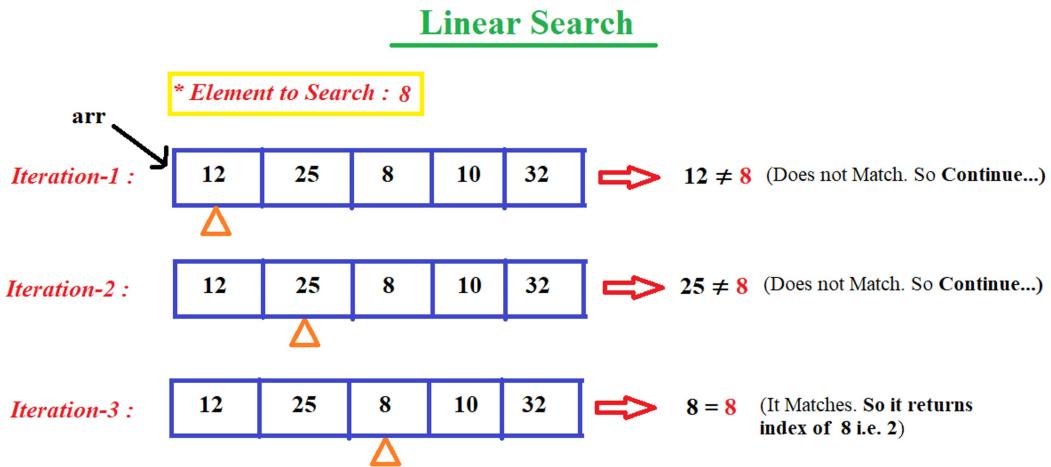
## ➤ Searching

It is a process of finding a desired element in a set(or list) of items.

## ➤ LINEAR SEARCH

It is a basic and simplest searching algorithm.

In linear search, we start searching from first element till the element we are looking for.



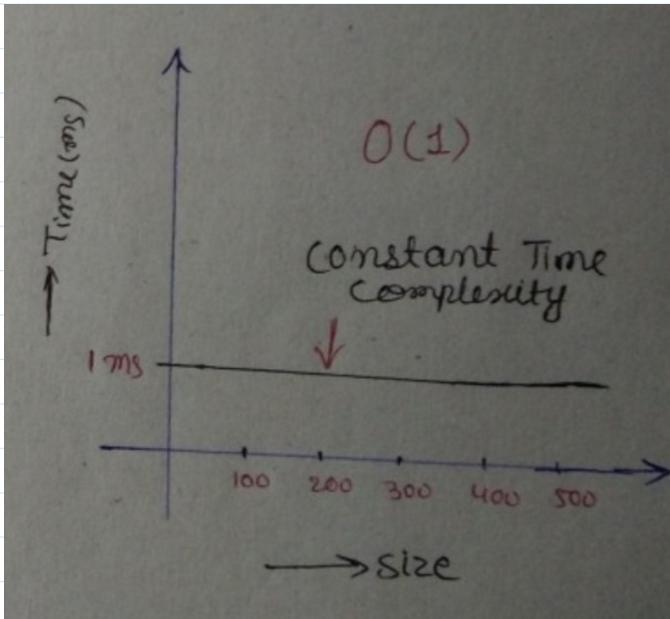
## ➤ Time complexity for linear search :

- Best case :  $O(1)$  //constant

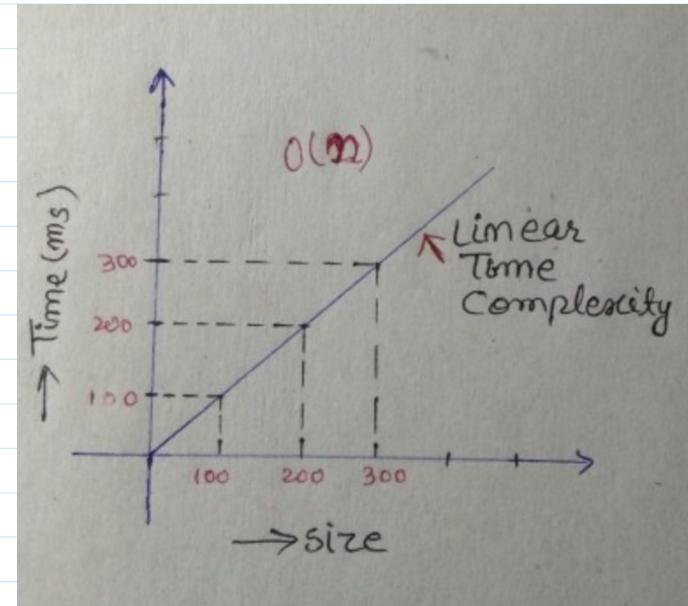
In linear search, best case occurs when the targeted(desired) value present at 0th index (1st location) means, for best case  $\rightarrow$  only one comparison made

- Worst case :  $O(n)$  //n=size of array

Worst case happens when the targeted (desired) element is compared will all the element of array one by one and it says element not found.  
i.e., we have done  $n$  comparisons (here,  $n \rightarrow$  size of array)



Best case



Worst case

//program

```

package com.array2;

import java.util.Scanner;

public class Linear_search {
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);

        int[] arr={2,4,9,66,77,88,99,101,121,131,147,158,196,210,224,225,693};
        int ser;

        System.out.println("enter the element you want to search");
        ser=in.nextInt();
        System.out.println(linear(arr,ser));

    }
    //search in the array : return the index if item found
    //other if item not found return -1
    static int linear(int[] arr,int target){
        if(arr.length == 0){
            return -1;
        }
        for (int index = 0; index < arr.length; index++) {
            if(arr[index]==target){
                return index;
            }
        }
    }
}

```

```
}

//this line will execute if none of the return statements above executed
// hence the target not found
return -1;

}

}
```

# BINARY SEARCH

22 January 2022 01:47

## Binary search:

- It is the most important algorithm.
- It is used for sorted array (either in ascending or descending order)

### ➤ Algorithm (when elements are in Ascending order) :

- Step 1 : Find the Middle element ( $\text{mid} = [\text{start} + \text{end}] / 2$ )
- Step 2 : check
  - If  $\text{target} > \text{middle element} \Rightarrow$  search in right side  
 $\text{start} = \text{mid} + 1$
  - else  $\Rightarrow$  search in left  
 $\text{end} = \text{mid} - 1$
- Step 3: if  $\text{target} == \text{middle}$   $\Rightarrow$  element found

### ➤ EXAMPLE:

$\text{arr} = [2, 4, 6, 9, 11, 12, 14, 20, 36, 48]$

0 1 2 3 4 5 6 7 8 9

mid

Target = 36

↑  
Ascending order

Step 1: Find middle element --->

$\text{mid} = [\text{start} + \text{end}] / 2 \Rightarrow [0+9]/2 \Rightarrow 04 \Rightarrow$  element at index 4 is mid element

Step 2: check --->

Here, target > mid ( $36 > 11$ )  $\Rightarrow$  check in right side

Now,

$\text{arr} = [2, 4, 6, 9, 11, 12, 14, 20, 36, 48]$

0 1 2 3 4 5 6 7 8 9

mid

$\text{Start} = \text{mid} + 1 = 4 + 1 = 5$

$\text{Mid} = (5 + 9)/2 = 7$

$\Rightarrow$  element at index 7 is mid element

Step 3: check:

Here, Target > mid ( $36 > 20$ )  $\Rightarrow$  check in right side

Now,

$\text{arr} = [2, 4, 6, 9, 11, 12, 14, 20, 36, 48]$

0 1 2 3 4 5 6 7 8 9

$$\begin{aligned} \text{Start} &= \text{mid} + 1 = 7 + 1 = 8 \\ \text{Mid} &= (8 + 9)/2 = 8 \\ &\Rightarrow \text{element at index 8 is mid element} \end{aligned}$$

Step 4: check:

Here, Target == mid ( $36 == 36$ )  $\Rightarrow$  element found at index 8.

### Algorithm (When elements are in descending Order):

Step 1. Find the middle element.

Step 2. check

if  $>$  mid  $\Rightarrow$  search in left side

End = mid - 1

Else  $\Rightarrow$  search in Right side

Start = mid + 1

Step 3. if target == middle  $\Rightarrow$  Element found

//program

```
package com.array2;
```

```
public class BinarySearch {
    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 9, 11, 12, 14, 20, 36, 48};
        int target = 4;
        System.out.println(binarySearch(arr, target));
```

7

```

    }
    static int binarySearch(int[] arr, int target) {
        int start = 0;
        int end = arr.length - 1;

        while (start <= end) {
            // find the middle element
            // int mid = (start + end) / 2; -- might be possible that (start + end) exceeds the range of integer
            int mid = start + (end - start) / 2; ----->
            if (target < arr[mid]) {
                end = mid - 1;
            } else if (target > arr[mid]) {
                start = mid + 1;
            } else {
                // ans found
                return mid;
            }
        }
        return -1;
    }
}
```

*//Output = 1*

$$\rightarrow \text{mid} = \frac{(S+E)}{2}$$

+ better to use to find mid

$\rightarrow \text{mid} = \frac{(S+E)}{2}$  This may Exceeds the int range

If we find for large Nos.

$$\text{int mid} = S + \frac{(E-S)}{2}$$

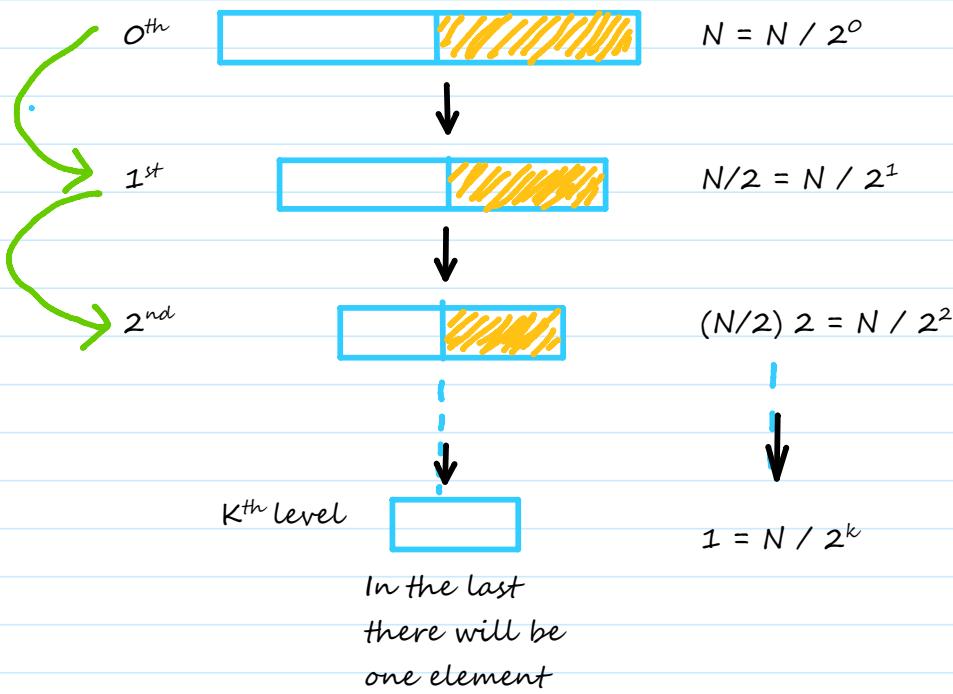
$$\left\{ \frac{S+E}{2}$$

$$\frac{S+E}{2}$$

} //Output =1

### > Time complexity

- Best case :  $O(1)$  //constant  
[when first middle element == target value]
- Worst case :  $O(\log n)$  //n -> size of array
- Find the maximum numbers of comparisons in the worst case.....



$$N / 2^k = 1$$

$$\begin{aligned} N &= 2^k \\ \log N &= \log(2^k) \\ K &= \log N / \log 2 \\ K &= \log_2 N \end{aligned}$$

← Size of array

Total no. of Comparisons in worst case

### > Why binary Search?

Suppose we have to find an element in an array of size 1000000 (1 million)

- In worst case,
  - Linear search will make 1 million comparisons
- In worst case for same array,
  - Binary search will make  $\log(1000000)$

i.e., 20 comparisons only

### > Order - Agnostic Binary Search :

- We know, for binary search we need sorted array but, Let's say, if we don't know the array is sorted in ascending or descending order.
- If  $\text{start} > \text{end} \Rightarrow$  Descending order
- If  $\text{start} < \text{end} \leq$  Ascending order

//program

```
package com.array2;

public class OrderAgnosticBS {
    public static void main(String[] args) {
        int[] arr= {2, 4, 6, 9, 11, 12, 14, 20, 36, 48};
        int[] arr2={88,55,22,10,6,3,2,1,1};

        System.out.println(find(arr,12));
        System.out.println(find(arr2,55));
    }

    static int find(int[] arr, int target){
        int start=arr[0];
        int end=arr[arr.length-1];
        if(start>end){
            return binarySearchDes(arr,target);
        }
        if(start<end){
            return binarySearchAes(arr,target);
        }
        return -1;
    }

    static int binarySearchAes(int[] arr, int target){
        int start=0;
        int end=arr.length-1;

        while (start<=end){
            //find the middle element
            //int mid = (start + end) / 2; --> might be possible that (start + end) exceeds the range of
            integer
            int mid = start + (end- start)/2;
            if(target<arr[mid]){
                end=mid-1;
            }else if(target>arr[mid]){
                start=mid+1;
            }else {
                //ans found
                return mid;
            }
        }
    }
}
```

```
        }
    }
    return -1;

}

static int binarySearchDes(int[] arr, int target){
    int start=0;
    int end=arr.length-1;

    while (start<=end){
        //find the middle element
        //int mid = (start + end) / 2; --> might be possible that (start + end) exceeds the range of
        integer
        int mid = start + (end- start)/2;
        if(target<arr[mid]){
            start=mid+1;
        }else if(target>arr[mid]){
            end=mid-1;
        }else {
            //ans found
            return mid;
        }
    }
    return -1;
}

//output:
5
1
```

?Sorts of question BS

08 February 2022 02:49

Use Binary search where you find sorted array

### Question 01 : Ceiling

arr = [2, 3, 5, 9, 14, 16, 18]

ceiling = smallest element in array greater or equal to target

ceiling(arr,target = 14) = 14

ceiling(arr,target = 15) = 16

arr = [2, 3, 5, 9, 14, 16, 18]



ceiling(arr,target = 4) = 9

ceiling(arr,target = 9) = 9

//program

package com.array2;

```
public class Ceiling {  
    public static void main(String[] args) {  
        int[] arr = {2, 3, 5, 9, 14, 16, 18};  
        int target = 4;  
        System.out.println("at Index: " + ceiling(arr, target));  
    }  
}
```

// ceiling = return the smallest element in array greater or equal to target

static int ceiling(int[] arr, int target) {

int start = 0;

int end = arr.length - 1;

while (start <= end) {

// find the middle element

// int mid = (start + end) / 2; --> might be possible that (start + end) exceeds the range of integer

int mid = start + (end - start) / 2;

if (target < arr[mid]) {

end = mid - 1;

```

}else if(target>arr[mid]){
    start=mid+1;
}else {
    //ans found
    return mid;
}
return start;

}
}

```

### Question 02 : Floor

Floor = return the greatest number  $\leq$  target

`floor(arr,target = 15) = 14`

`arr = [2, 3, 5, 9, 14, 16, 18]`

0 1 2 3 4 5 6

S M E

Step 1

S M E

Step 2

S  
M  
E

Step 3

E S

return end

//sample program

package com.array2;

public class Floor {

public static void main(String[] args) {

int[] arr= {2, 3, 5, 9, 14, 16, 18};

int target=15;

System.out.println("at index: "+floor(arr,target));

}

//floor = return the greatest number  $\leq$  target

static int floor(int[] arr, int target){

int start=0;

int end=arr.length-1;

while (start<=end){

```

//find the middle element
//int mid = (start + end) / 2; --> might be possible that (start + end) exceeds the range of integer
int mid = start + (end - start)/2;
if(target<arr[mid]){
    end=mid-1;

}else if(target>arr[mid]){
    start=mid+1;
}else {
    //ans found
    return mid;
}
return end;
}
}

```

### 3. Find Smallest Letter Greater Than Target

Given a characters array letters that is sorted in non-decreasing order and a character target, return the smallest character in the array that is larger than target. Note that the letters wrap around.

- For example, if target == 'z' and letters == ['a', 'b'], the answer is 'a'.

Example 1:

Input: letters = ["c", "f", "j"], target = "a"  
Output: "c"

Example 2:

Input: letters = ["c", "f", "j"], target = "c"  
Output: "f"

Example 3:

Input: letters = ["c", "f", "j"], target = "d"  
Output: "f"

//program

package com.array2;

```

public class NextGreatestLetter {
    public static void main(String[] args) {
        char[] letters = {'c', 'f', 'j'};
        char target = 'c';
        System.out.println(nextGreatestLetter(letters, target));
    }
    static char nextGreatestLetter(char[] letters, char target){
        int start=0;
        int end=letters.length-1;
        while (start<=end){
            //find the middle element
            //int mid = (start + end) / 2; --> might be possible that (start + end) exceeds the range of integer
            int mid = start + (end - start)/2;
            if(target<letters[mid]){
                end=mid-1;
            }
            else {
                start=mid+1;
            }
        }
    }
}

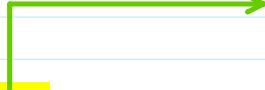
```

→ Start % letters.length becoz as per the guideline, we

```

        }
    } else {
        start=mid+1;
    }
    return letters[start%letters.length];
}
}

```



Start % letters.length becoz as per the guideline, we have to return 0th index character if the targeted is greater than given array so in this incase we can use start % letters.length suppose -->

1. Input: letters = ["c", "f", "j"], target = "d"  
Output: "f" (1 % 3 = 0)

2. For example, if target == 'z' and letters == ['a', 'b'], the answer is 'a'. (Here 2%2 = 0)

#### 4. Find First and Last Position of Element in Sorted Array

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with  $O(\log n)$  runtime complexity.

**Example 1:**

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

**Example 2:**

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

**Example 3:**

Input: nums = [], target = 0

Output: [-1,-1]

```

class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] ans={-1,-1};
        for(int i=0; i<nums.length; i++){
            if(nums[i]==target){
                ans[0] = i;
                break;
            }
        }
        for(int i=nums.length-1; i>=0; i--){
            if(nums[i]==target){
                ans[1] = i;
                break;
            }
        }
        return ans;
    }
}

```

#### 5. Find position of an element in a sorted array of infinite numbers

Suppose you have a sorted array of infinite numbers, how would you search an

element in the array?

Source: Amazon Interview Experience.

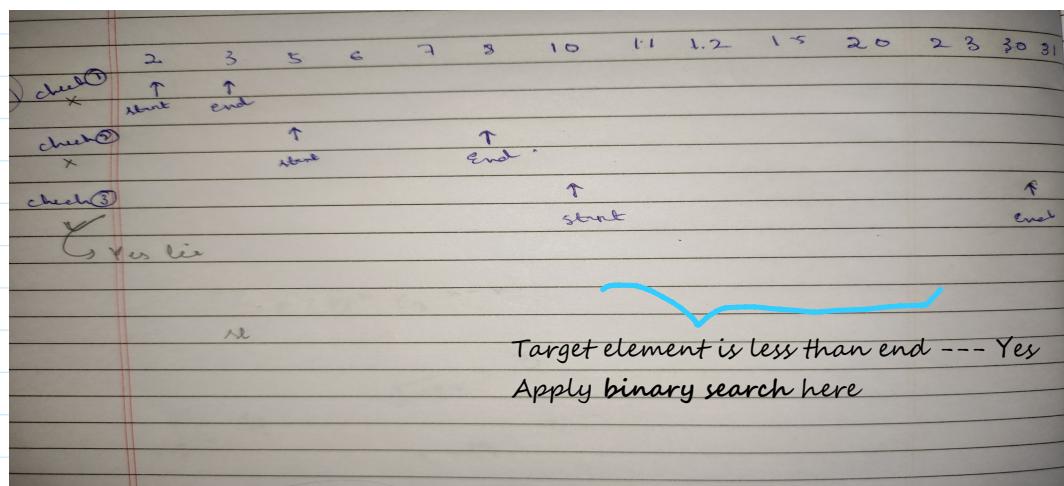
Since array is sorted, the first thing clicks into mind is binary search, but the problem here is that we don't know size of array.

If the array is infinite, that means we don't have proper bounds to apply binary search. So in order to find position of key, first we find bounds and then apply binary search algorithm.

Let low be pointing to 1st element and high pointing to 2nd element of array. Now compare key with high index element,

->if it is greater than high index element then copy high index in low index and double the high index.

->if it is smaller, then apply binary search on high and low indices found.



```
package com.array2;

public class Infinite_sorted_array {
    public static void main(String[] args) {
        int[] arr ={3,5,7,9,10,90,100,130,140,160,170};
        int target=10;
        System.out.println(ans(arr,target));

    }
    static int ans(int arr[], int target){
        //start with the box of size 2
        //first start with a box of size 2:
        int start=0;
        int end =1;
        while(target>arr[end]){
            int temp = end+1; //this is my new start

            //double the box value
            //end = previous end + sizeofbox*2
            //size of box = end - start +1

            end= end+(end - start +1)*2;
            start=temp;

        }
        return binarySearch(arr,target,start,end);
    }
}
```

```

    }
    static int binarySearch(int[] arr, int target, int start, int end){
        // int start=0;
        // int end=arr.length-1;

        while (start <= end){
            //find the middle element
            //int mid = (start + end) / 2; --> might be possible that (start + end) exceeds the range of
            integer
            int mid = start + (end - start) / 2;
            if(target < arr[mid]){
                end=mid-1;

            }else if(target>arr[mid]){
                start=mid+1;
            }else {
                //ans found
                return mid;
            }
        }
        return -1;
    }
}

```

//problem is while doubling box index it can go out length of the array which it will throw array out of index

Example -- {2,3,5,6,7,8,10,11,12,15,20,23,30}

Here suppose target element is 15

Check 1 - start = 0, end = 1

Check 2 - start = 2, end = 5

Check 3 - start = 6, end = **13**

Here we can see as per the algorithm end is 13, but problem is total element in given array is 12. therefore it will throw array out of index

## 1. Peak Index in a Mountain Array

Let's call an array arr a mountain if the following properties hold:

- arr.length >= 3
- There exists some  $i$  with  $0 < i < \text{arr.length} - 1$  such that:
  - $\text{arr}[0] < \text{arr}[1] < \dots < \text{arr}[i-1] < \text{arr}[i]$
  - $\text{arr}[i] > \text{arr}[i+1] > \dots > \text{arr}[\text{arr.length} - 1]$

Given an integer array arr that is guaranteed to be a mountain, return any  $i$  such that  $\text{arr}[0] < \text{arr}[1] < \dots < \text{arr}[i - 1] < \text{arr}[i] > \text{arr}[i + 1] > \dots > \text{arr}[\text{arr.length} - 1]$ .

Example 1:

Input: arr = [0,1,0]

Output: 1

Example 2:

Input: arr = [0,2,1,0]

Output: 1

Example 3:

Input: arr = [0,10,5,2]

Output: 1

```

package com.array2;

import java.util.Arrays;

public class PeakIndex {
    public static void main(String[] args) {
        int[] arr = {3,4,5,1};
        System.out.println(peakIndexInMountainArray(arr));
    }

    static int peakIndexInMountainArray(int[] arr) {
        int ans = 0, temp, start = 0, end = arr.length-1;
        while (start < end) {
            int mid = start+(end-start)/2;
            if(arr[mid] > arr[mid+1]){
                //you are in dec part of array
                //this may be the ans, but you look left
                //this is why end!= mid-1
                end=mid;
            }else{
                //you are asc part of the array
                start=mid+1; //because we know that mid+1 element > mid element
            }
        }
        //in the end, start == end and pointing to the largest number because of the 2 checks above 3
        //start and end are always trying to find max element in the above 2 checks
        // hence, when they are pointing to just one element, that is the max one because that is what the
        check say
        // more elaboration : at every point of time for start and end, they have the best possible answer
        till that time.
        // and if we are saying that only item is remaining , hence cuz of the above line is the best
        possible ans.
        return end; //return end or start
    }
}

```

## 1. Find in Mountain Array

You may recall that an array arr is a mountain array if and only if:

- o arr.length  $\geq 3$
- o There exists some i with  $0 < i < \text{arr.length} - 1$  such that:
  - $\text{arr}[0] < \text{arr}[1] < \dots < \text{arr}[i - 1] < \text{arr}[i]$
  - $\text{arr}[i] > \text{arr}[i + 1] > \dots > \text{arr}[\text{arr.length} - 1]$

Given a mountain array mountainArr, return the minimum index such that  $\text{mountainArr.get(index)} == \text{target}$ . If such an index does not exist, return -1.  
 You cannot access the mountain array directly. You may only access the array using a MountainArray interface:

- o `MountainArray.get(k)` returns the element of the array at index k (0-indexed).
- o `MountainArray.length()` returns the length of the array.

Submissions making more than 100 calls to `MountainArray.get` will be judged Wrong Answer. Also, any solutions that attempt to circumvent the judge will result in

disqualification.

Example 1:

Input: array = [1,2,3,4,5,3,1], target = 3

Output: 2

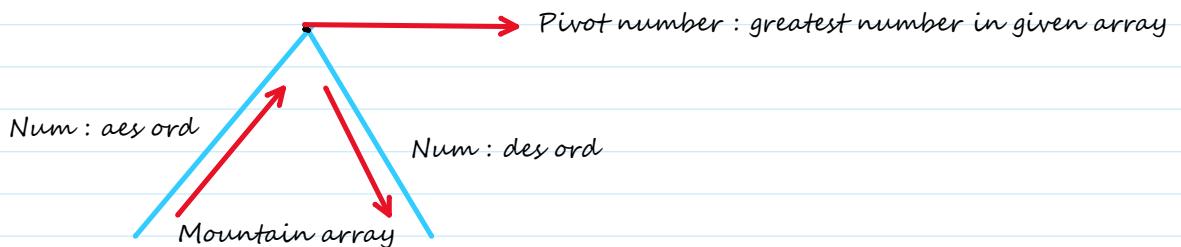
Explanation: 3 exists in the array, at index=2 and index=5. Return the minimum index, which is 2.

Example 2:

Input: array = [0,1,2,4,2,1], target = 3

Output: -1

Explanation: 3 does not exist in the array, so we return -1.



```
package com.array2;
```

```
public class FindInMountainArray {  
    public static void main(String[] args) {  
        int[] array = {1,12,13,14,15,3,0};  
        int target = 3;  
        System.out.println(findInMountainArray(array,target));  
    }  
    static int findInMountainArray(int[] arr, int target){  
        //check if the targeted element is there or not before pivot num if not then it checks after the pivot  
        num  
        int i = index(arr);  
        if (binarySearchAes(arr,target,0,i) != -1){  
            return binarySearchAes(arr,target,0,i);  
        }else if(binarySearchDes(arr,target,i,arr.length-1) != -1){  
            return binarySearchDes(arr,target,i,arr.length-1);  
        }  
        return -1;  
    }  
    static int index(int[] arr){  
        //gives pivot num in the given array (pivot point : largest no)  
        int start=0;  
        int end=arr.length-1;  
        while(start<end){  
            int mid = start + (end-start)/2;  
            if(arr[mid]>arr[mid+1]){  
                end = mid;  
            }else {  
                start = mid + 1;  
            }  
        }  
        return end;  
    }  
    static int binarySearchAes(int[] arr, int target,int start, int end){  
        while (start<=end){  
            int mid = start + (end- start)/2;
```

```

if(target<arr[mid]){
    end=mid-1;

}else if(target>arr[mid]){
    start=mid+1;
}else {
    //ans found
    return mid;
}

}

return -1;
}

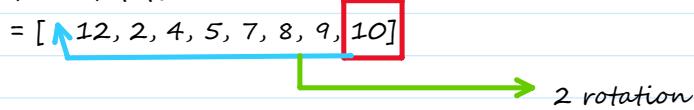
static int binarySearchDes(int[] arr, int target, int start, int end){
    while (start<=end){
        int mid = start + (end- start)/2;
        if(target<arr[mid]){
            start=mid+1;
        }else if(target>arr[mid]){
            end=mid-1;
        }else {
            //ans found
            return mid;
        }
    }
    return -1;
}
}

```

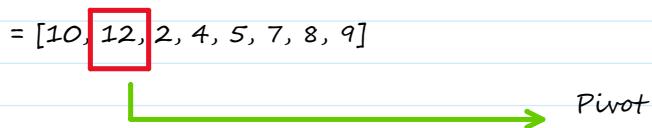
### 8. Search in Rotated Sorted Array



After one rotation :

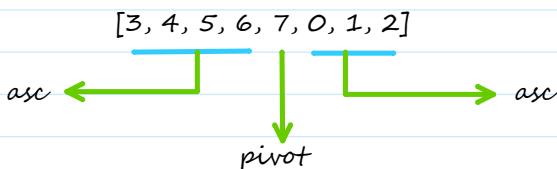


After 2nd rotation :



- Find the pivot in the array

Pivot => from where your next has ascending order.



- > find pivot
- > search in first half using BS (0, pivot)
- > otherwise, search in second half (pivot +1, end)

```

package com.array2;

public class RotatedSortedArray {
    public static void main(String[] args) {
        int[] nums = {10,11,12,13,14,15,0,1,2};
        int target = 2;
        System.out.println(search(nums,target));
    }
    static int search(int[] nums, int target) {
        int ind = index(nums);
        if(binarySearchAes(nums,target,0,ind) != -1){
            return binarySearchAes(nums,target,0,ind);
        }else if(binarySearchDes(nums, target,ind,nums.length-1) != -1){
            return binarySearchDes(nums, target,ind,nums.length-1);
        }
        return -1;
    }
    static int index(int[] arr){

        int start=0;
        int end=arr.length-1;
        while(start<end){
            int mid = start + (end-start)/2;
            if(arr[mid]>arr[mid+1]){
                end = mid;
            }else {
                start = mid + 1;
            }
        }
        return end;
    }
    static int binarySearchAes(int[] arr, int target, int start, int end){
        // int start=0;
        // int end=arr.length-1;

        while (start<=end){
            //find the middle element
            //int mid = (start + end) / 2; --> might be possible that (start +
            end) exceeds the range of integer
            int mid = start + (end- start)/2;
            if(target>arr[mid]){
                end=mid-1;

            }else if(target>arr[mid]){
                start=mid+1;
            }else {
                //ans found
                return mid;
            }
        }
        return -1;
    }
    static int binarySearchDes(int[] arr, int target, int start, int end){
        while (start<=end){
            //find the middle element
            //int mid = (start + end) / 2; --> might be possible that (start +

```

end) exceeds the range of integer

```
int mid = start + (end - start)/2;
if(target<arr[mid]){
    start=mid+1;
}else if(target>arr[mid]){
    end=mid-1;
}else {
    //ans found
    return mid;
}
return -1;
```

}

# ? Binary search in 2d array

22 January 2022 02:22

# SORTING

07 March 2022 01:52

*Sorting :It is a process of arranging items systematically.*

## Bubble sort

07 March 2022 01:52

- It is simplest algorithm that works by repeatedly swapping the adjacent elements if they are in wrong orders



Bubble sort is also known as sinking sort or exchange sort.

- Let's understand more deeply how actually bubble sort works



- Complexity :

- Space complexity :  $O(1)$  //constant

- Since here no extra space is required i.e., like copying the array etc. is not required.

Also known as inplace sorting algorithm

- Time Complexity :

- 1. Best case: Array is sorted.

$i = 0$   
First pass

1, 2, 3, 4, 5  
~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~

Only once it ran we don't need to check it again

$N \rightarrow$  no of comparisons

If array size is 10

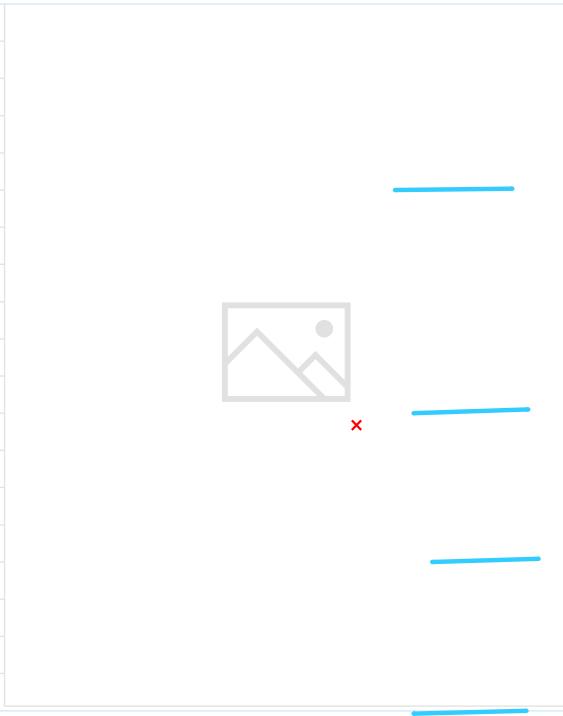
Then in best case it will take 10 no of comparisions only

NOTE : when j never swaps for value of 1, it means array is sorted. Hence, you can end the program

• Best case comparisons =  $N - 1 \Rightarrow N$  //In time complexity constant are ignored

Since in time complexity constants are ignored, we don't want exact time, we just want relationship i.e., mathematical function.

2. Worst case : Sorting descending order array to ascending order.



$$\text{Total comparisions} = (N-1) + (N-2) + (N-3) + (N-4)$$

$$\text{Comparisions} = 4N - (1 + 2 + 3 + 4)$$

$$4n - (N * (N+1)) / 2$$

$$8N - (N^2 + N) / 2$$

$$(8N - N^2 - N) / 2$$

$O(7N - N^2) / 2$  // In big - o - notation constants are cancelled and least dominating terms are removed

$$O(N^2)$$

- Stable sorting algorithm : 10 20 20 30 10



Stable sorting algorithm basically means, when original order is maintained for values that are equal.

- Unstable sorting algorithm : 10 20 20 30 10



Unstable sorting algorithm means, When original order is not maintained where two or more values are same

```
// program
package com.Rec4;

import java.util.Arrays;

public class Temp {
    public static void main(String[] args) {
        int[] arr ={2, 10, 11, 12, 66, 96};
```

```
bubbleSort(arr);
}
static void bubbleSort(int[] arr){
    boolean swapped=false;
    for(int i =1; i<arr.length-1; i++){
        for (int j = 0; j < arr.length-i; j++) {
            if(arr[j]>arr[j+1]){
                swap(arr,j,j+1);
                swapped=true;
            }
        }
    }
    //if you did not swap for a particular value of i, it means the array is sorted hence stop the program
    if(!swapped) return;
    System.out.println(Arrays.toString(arr));
}

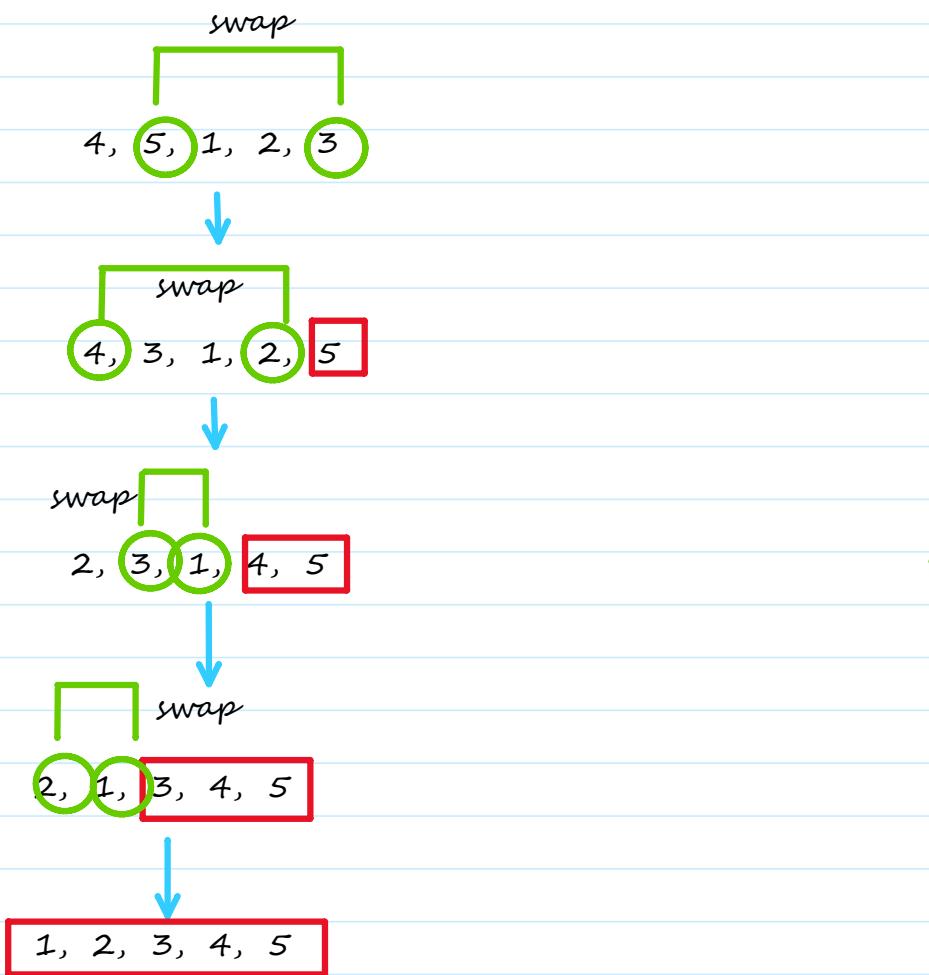
static void swap(int[]arr, int j, int j1) {
    int temp = arr[j];
    arr[j]=arr[j1];
    arr[j1]=temp;
}

}
```

# Selection Sort

12 March 2022 16:07

Selection Sort: select an element & put it on its correct index



Here we selected maximum element and put it on right index, we can do vice versa i.e., select minimum element and put it on right index

## ➤ Complexity

Total comparisons => 4, 5, 1, 2, 3

(n-1)

4, 3, 1, 2, 5

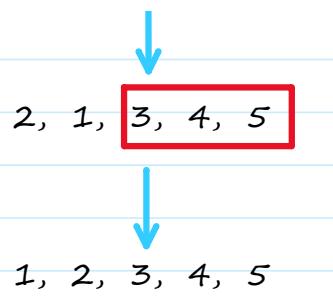
(n-2)

2, 3, 1, 4, 5

(n-3)

Already at correct position ignore in future steps

.. 11



$$0 + 1 + 2 + 3 + \dots + (n-1) = n(n-1)/2 = (n^2 - n)/2$$

Neglect the dominating and constant term

- Worst case  $\rightarrow O(n^2)$
- Best case  $\rightarrow O(n^2)$
- It is not stable sorting algorithm.
- It performs well on small lists

```

package com.array2;

import java.util.Arrays;

public class SelectionSort {
    public static void main(String[] args) {
        int[] arr = {5, 3, 4, 1, 2};
        selection(arr);
        System.out.println(Arrays.toString(arr));
    }

    static void selection(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            // find the max item in the remaining array and swap with correct index
            int last = arr.length - i - 1;
            int maxIndex = getMaxIndex(arr, 0, last);
            swap(arr, maxIndex, last);
        }
    }

    static void swap(int[] arr, int first, int second) {
        int temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
    }
}

```

```
    arr[second] = temp;
}

static int getMaxIndex(int[] arr, int start, int end) {
    int max = start;
    for (int i = start; i <= end; i++) {
        if (arr[max] < arr[i]) {
            max = i;
        }
    }
    return max;
}
```

# ? Insertion Sort

14 March 2022 23:57

Array => 5, 3, 4, 1, 2

[partially sorting the

,

# ? Cycle Sort

10 May 2022 00:47

# String

15 March 2022 00:01

- **Stack Memory** : - When we declare a variable

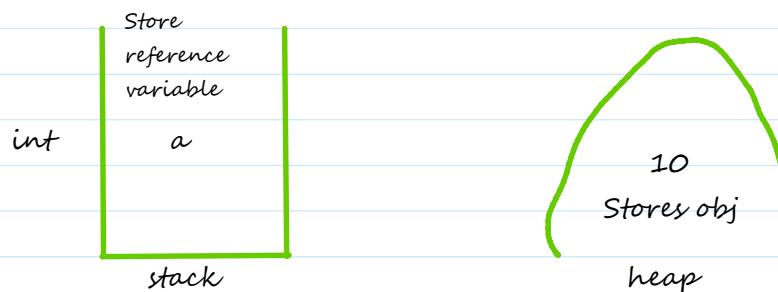
Eg :- int a = 10;

So, here the reference variable is stored in stack memory

- **Heap Memory** : Reference variable stored in stack memory is pointing to the object of that variable are stored in heap memory

int a = 10

S H



- **Memory allocation** : Memory allocation specifies the memory address to a program.

There are two types of memory

- A) stack memory
- B) Heap memory

- **Static memory allocation** :

- it performs type checking at compile time.
- Here, errors will show at compile time
- Declare data type before you use it
- More control & runtime errors are reduced

- **Dynamic memory allocation** : - perform type checking at runtime

- Here, errors might not shown at till the program is run
- No need to declare a datatype of a variable
- It saves time in writing code but might give error in runtime

- **Garbage collection** :-

- More than one reference variable can points to the same object.
- If any change made of in the object of an reference variable that will be reflected to all others pointing to the same object.
- If there is an object without reference variable then the object will be destroyed by garbage collection
- So that's how garbage collection works.

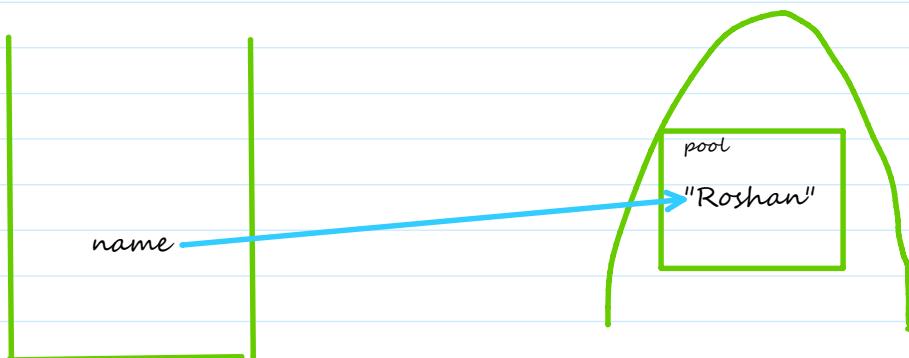


String name = "Roshan Prusty"

Concept -

- a. String pool
- b. Immutability

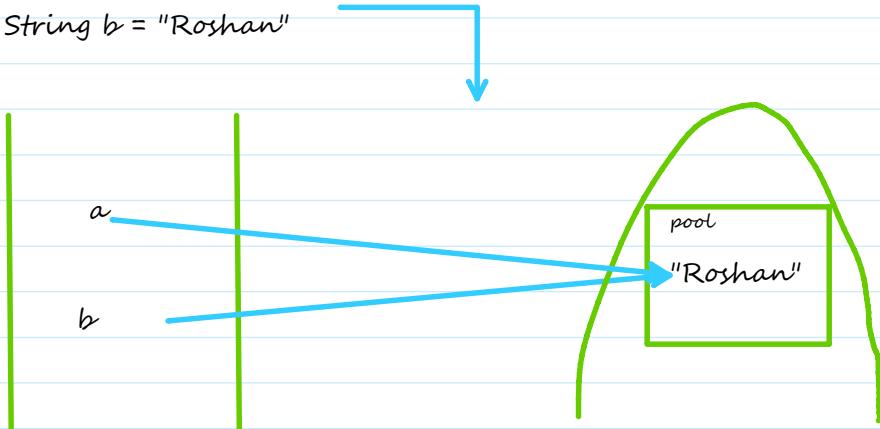
- String Pool - String pool is a separate memory structure inside heap memory



Q. Why we have string pool like concepts in String

Ans. Because similar type objects will be not required to recreate

- String a = "Roshan"
- String b = "Roshan"



Q. As in above diagram 2 reference variable is pointing element is pointing to same elements than if one changed than other will also change?

Ans- No, because you cannot change the object  
It will just create new object.

```
package com.string;
```

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

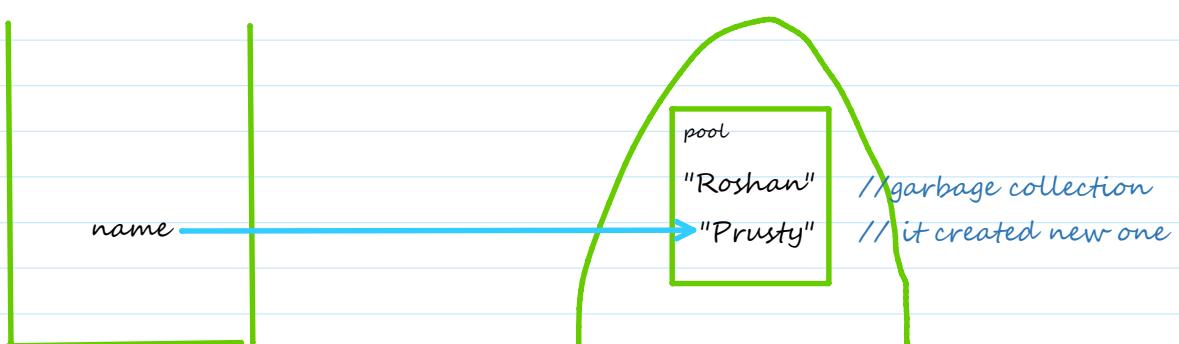
```

String name = "roshan";
System.out.println(name);
name = "prusty";
System.out.println(name);

}

//roshan
//prusty

```



Q. Why we can't modify string objects?

Ans. Due to security reason.....Think there is passwords of same object than any one reference variable will change the object than for every reference variable object will changed.



#### ➤ Comparisons of string:

1. `==` method



`A -----> "Roshan"`  
`B -----> "Roshan"`

`A == B will give false`

`A -----> "Roshan"`  
`B -----> "Roshan"`

`A == B will give true`

## ➤ How to create different object same value

```
String a = new String ("Roshan")
String b = new String ("Roshan")
// creating these value outside the pool but in heap
```

```
String s1= "welcome";
String s2= "welcome";
```



```
a == b //false
s1 == s2 //true
```

```
package com.string;

import java.util.Arrays;

public class St {
    public static void main(String[] args) {

        String name1 = new String("Roshan");
        String name2 = new String("Roshan");
        System.out.println(name1==name2);      //false
        System.out.println(name1.equals(name2)); //true

        String a ="hello";
        String b ="hello";
        System.out.println(a==b);              //true
        System.out.println(a.equals(b));       //true
        System.out.println(a.charAt(0));       //h
    }
}
```

- why we don't have string pool type concept in other data type in java ?

String Pool is possible only because String is immutable in Java and its implementation of String interning concept. String pool is also example of Flyweight design pattern.

String pool helps in saving a lot of space for Java Runtime although it takes more time to create the String.

When we use double quotes to create a String, it first looks for String with the same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

However using new operator, we force String class to create a new String object in heap space. We can use intern() method to put it into the pool or refer to another String object from the string pool having the same value.

## Substring & Subsequence

01 June 2022 15:46

### Substring

The `substring(int beginIndex, int endIndex)` method of the `String` class. It returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex - beginIndex`.

```
package com.string;

import java.util.Scanner;

public class Substring {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String name = in.nextLine();
        System.out.println(name);
        System.out.println("Substring of given name is " + name.substring(0,5));
    }
}
```

i/p --- roshan prusty  
o/p-  
roshan prusty  
Substring of given name is rosha

roshan prusty  
0 1 2 3 4 5 6 7 8 9 10 11 12

//print starting index till (last index -1)

### Subsequence

This method returns a new character sequence that is a subsequence of this sequence.

```
package com.string;

public class Subsequence {
    public static void main(String[] args) {
        String Str = new String("Welcome to world");

        System.out.print("Return Value :");
        System.out.println(Str.subSequence(0, 10));

        System.out.print("Return Value :");
        System.out.println(Str.subSequence(10, 15));
    }
}
```

### Output -

Return Value :Welcome to  
Return Value : worl

**What is the difference between substring vs subsequence?**

A Substring takes out characters from a string placed between two specified indices in a continuous order. On the other hand, subsequence can be derived from another sequence by deleting some or none of the elements in between but always maintaining the relative order of elements in the original sequence.

**What is a subsequence of a string?**

A subsequence of a given string is generated by deleting some or no character of a given string without changing the order of the remaining elements. It can contain consecutive elements that were not consecutive in the original sequence and an empty string. Examples: Input: "apb"  
Output: "a," "p," "b," "ap," "pb," "ab," "apb," and. ""

**What is a substring in a string?**

A substring is a contiguous sequence of characters within a string. For example, "is a rainy" is a substring of "Today is a rainy day."

## Points to remember

26 August 2022 12:20

- Use `.equals(str)` rather than `==`

Use the `equals()` method to check if 2 strings are the same. The `equals()` method is case-sensitive, meaning that the string "HELLO" is considered to be different from the string "hello". The `==` operator does not work reliably with strings. Use `==` to compare primitive values such as int and char. Unfortunately, it's easy to accidentally use `==` to compare strings, but it will not work reliably. Remember: use `equals()` to compare strings. There is a variant of `equals()` called `equalsIgnoreCase()` that compares two strings, ignoring uppercase/lowercase differences.

```
String a = "hello";
String b = "there";
if (a.equals("hello")) {
    // Correct -- use .equals() to compare Strings
}
if (a == "hello") {
    // NO NO NO -- do not use == with Strings
}
// a.equals(b) -> false
// b.equals("there") -> true
// b.equals("There") -> false
// b.equalsIgnoreCase("THERE") -> true
```

- The java string `valueOf()` method converts different types of values into string. By the help of `string valueOf()` method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

For more details do visit :<https://www.javatpoint.com/java-string-valueof>

- When to use `(String)` vs when to use `string valueOf()`
- StringBuffer

# Pattern Question

23 March 2022 23:47

## ➤ How to approach

- No of lines = no of rows = no of times it will run in outer loop
- Identify for every row number, how many column are there or types of element in column.
- What do you need to print.

# Questions

24 March 2022 02:09

1. \*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

2. \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

3. \*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

4. 1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

5. \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

6. \*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

7. \*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

8. \*

\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

9. \*\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*

10. \*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

11. \* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

12. \* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

13. \*

\* \*

\* \*

\* \*

\*\*\*\*\*

14. \*\*\*\*\*

\* \*  
\* \*  
\* \*  
\*

15. \*

\* \*  
\* \*  
\* \*  
\* \*  
\* \*  
\* \*  
\* \*  
\*

16. 1

1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1

17. 1

212

32123

4321234

32123

212

1

18. \*\*\*\*\*

\*\*\*\* \*\*\*\*

\*\*\* \*\*\*

\*\* \*\*

\* \*

\* \*

\*\* \*\*

\*\*\* \*\*\*

\*\*\*\* \*\*\*\*

\*\*\*\*\*

19. \* \*

\*\* \*\*

\*\*\* \*\*\*

\*\*\*\* \*\*\*\*

\*\*\*\*\*

\*\*\*\* \*\*\*\*

\*\*\* \*\*\*

\*\* \*\*

\* \*

20. \*\*\*\*

\* \*

\* \*

\* \*

\*\*\*\*

21. 1

2 3

4 5 6

7 8 9 10  
11 12 13 14 15

22. 1  
0 1  
1 0 1  
0 1 0 1  
1 0 1 0 1

23. \* \*  
\* \* \* \*  
\* \* \* \*

24. \* \*  
\*\* \*\*  
\* \* \* \*  
\* \*\* \*  
\* \*\* \*  
\* \* \* \*  
\* \* \* \*  
\*\* \*\*  
\* \*

25. \*\*\*\*\*  
\* \*  
\* \*  
\* \*  
\*\*\*\*\*

26. 1 1 1 1 1 1  
2 2 2 2 2  
3 3 3 3  
4 4 4

5 5

6

27. 1 2 3 4 17 18 19 20  
5 6 7 14 15 16  
8 9 12 13  
10 11

28. \*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\*

\*

29.

\* \*

\*\* \*\*

\*\*\* \*\*\*

\*\*\*\* \*\*\*\*

\*\*\*\*\* \*\*\*\*\*

\*\*\*\*\* \*\*\*\*

\*\*\* \*\*\*

\*\* \*\*

\* \*

30. 1

2 1 2

3 2 1 2 3

4 3 2 1 2 3 4

5 4 3 2 1 2 3 4 5

31. 4 4 4 4 4 4 4

4 3 3 3 3 3 4

4 3 2 2 2 3 4

4 3 2 1 2 3 4

4 3 2 2 2 3 4

4 3 3 3 3 3 4

4 4 4 4 4 4 4

32. E

D E

C D E

B C D E

A B C D E

\*\*\*\*\*

\*\*\*\*\*

9. \*\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*

10. \*

\* \*

\* \* \*

\* \* \* \*

\*\*\*\*\*

11. \* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

12. \* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

13. \*

\* \*

\* \*

\* \*

\*\*\*\*\*

14. \*\*\*\*\*

\* \*

\* \*

\* \*

\*

15. \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\*

16. 1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

17. 1

212

32123

4321234

32123

212

1

18. \*\*\*\*\*

\*\*\*\* \*\*\*\*

\*\*\* \*\*\*

\*\* \*\*

\* \*

\* \*

\*\* \*\*

\*\*\* \*\*\*  
\*\*\*\* \*\*\*\*  
\*\*\*\*\*

19. \* \*  
\*\* \*\*  
\*\*\* \*\*\*  
\*\*\*\* \*\*\*\*  
\*\*\*\*\*  
\*\*\*\* \*\*\*  
\*\*\* \*\*\*  
\*\* \*\*  
\* \*

20. \*\*\*\*  
\* \*  
\* \*  
\* \*  
\*\*\*\*

21. 1  
2 3  
4 5 6  
7 8 9 10  
11 12 13 14 15

22. 1  
0 1  
1 0 1  
0 1 0 1  
1 0 1 0 1

23. \* \*

\* \* \* \*

\* \* \*

24. \* \*

\*\* \*\*

\* \* \* \*

\* \* \* \*

\* \* \* \*

\* \* \* \*

\* \* \* \*

\* \* \* \*

\*\* \*\*

\* \*

25. \*\*\*\*\*

\* \*

\* \*

\* \*

\*\*\*\*\*

26. 1 1 1 1 1 1

2 2 2 2 2

3 3 3 3

4 4 4

5 5

6

27. 1 2 3 4 17 18 19 20

5 6 7 14 15 16

8 9 12 13

10 11

28. \*

\* \*  
\* \* \*  
\* \* \* \*  
\* \* \* \*  
\* \* \*  
\* \*  
\*

29.

\* \*  
\*\* \*\*  
\*\*\* \*\*\*  
\*\*\*\* \*\*\*\*  
\*\*\*\*\* \*\*\*\*\*  
\*\*\*\* \*\*\*\*  
\*\*\* \*\*\*  
\*\* \*\*  
\* \*

30. 1

2 1 2  
3 2 1 2 3  
4 3 2 1 2 3 4  
5 4 3 2 1 2 3 4 5

31. 4 4 4 4 4 4 4

4 3 3 3 3 3 4  
4 3 2 2 2 3 4  
4 3 2 1 2 3 4  
4 3 2 2 2 3 4  
4 3 3 3 3 3 4  
4 4 4 4 4 4 4

32. E

D E

C D E

B C D E

A B C D E

33. a

B c

D e F

g H i J

k L m N o

34. E D C B A

D C B A

C B A

B A

A

35. 1 1

12 21

123 321

12344321

## Recursion 1 : INTRODUCTION

31 March 2022 17:29

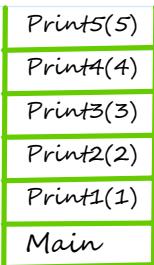
**Recursion :** Function that calls itself

```
package com.Recursion;

public class Rec {
    public static void main(String[] args) {
        print1(1);
    }

    static void print1(int n){
        System.out.println(n);
        print2(2);
    }
    static void print2(int n){
        System.out.println(n);
        print3(3);
    }
    static void print3(int n){
        System.out.println(n);
        print4(4);
    }
    static void print4(int n){
        System.out.println(n);
        print5(5);
    }
    static void print5(int n){
        System.out.println(n);
    }
}
```

- When the function is staying in the stack it means function call is currently going on.
- When function finishes Executing it is removed from the stack and flow of the program is restored to where the function is called.



```
package com.Recursion;
```

```
public class Rec2 {
    public static void main(String[] args) {
        System.out.println(print(1));
    }

    static int print(int num){
        //base condition
        if(num==5){
```

Base cond<sup>n</sup> in  
Recursion -

Base cond<sup>n</sup> in

Recursion -  
condition where the  
recursion will stop  
making calls

No base cond<sup>n</sup> -

Function call will  
keep happening in  
which stack  
memory will be  
filled then there  
Memory of computer  
will exceed the  
limit ---> it will  
throw a stack  
overflow error

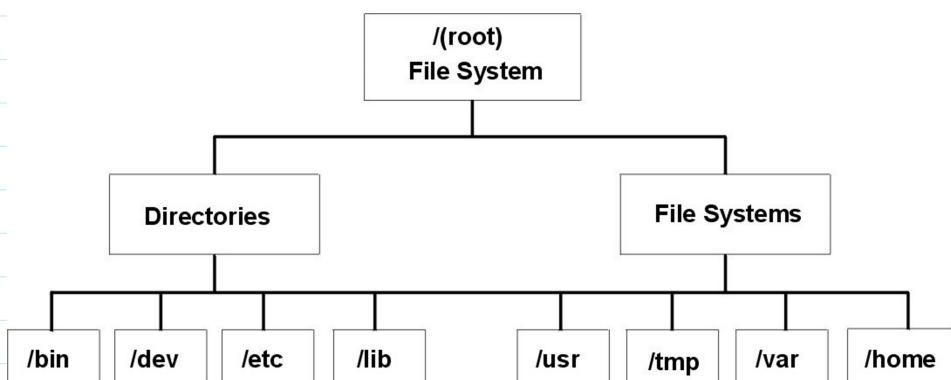
```
//base condition
if(num==5){
    return 5;
}
System.out.println(num);
return print(num+1); }
```

It calling itself but created  
another separate space for  
every value

- Why recursion?

- ❖ Recursion is made for solving problems that can be broken down into smaller, repetitive problems. It is especially good for working on things that have many possible branches and are too complex for an iterative approach.

One good example of this would be searching through a file system. You could start at the root folder, and then you could search through all the files and folders within that one. After that, you would enter each folder and search through each folder inside of that.



Recursion works well for this type of structure because you can search multiple branching paths without having to include many different checks and conditions for every possibility.

For those of you who are familiar with data structures, you might notice that the image above of the file system looks a lot like a tree structure.

Trees and graphs are another time when recursion is the best and easiest way to do traversal.

- ❖ Space complexity is not constant because of recursive calls.

- ❖ It helps us in breaking down bigger problem into smaller problem.

- Should I always use recursion?

Recursion seems really useful! Maybe I should use it all the time?

Well, like anything, recursion is best in doses. Recursion is a useful tool, but it can increase memory usage.

So let's go back to the factorial call stack image from above. Every time we add a new call to the stack, we are increasing the amount of memory that we are using. If we are analysing the algorithm using Big O notation, then we might note that this increases our space complexity.

There are times when we might want to pay this cost in order to get a short, useful algorithm, like when we are traversing a tree. But there are other times when there may be better, more efficient ways of solving this problem.

For many small projects, the call stack won't hamper your program very much. However, once your program starts making many recursive calls, then you might want to consider the potential impact of your large call stack.

Another thing to consider is that understanding and reading your code might sometimes be easier to do in an iterative solution.

Using recursion is great because it takes many of the incremental sub problems out of your hands.

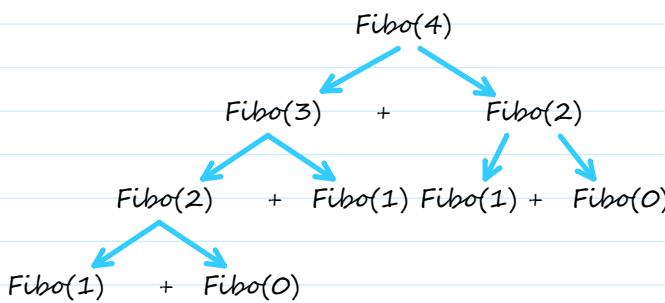
But when you are trying to fully understand the sub problems that you are solving and how they are being solved, it might be worth implementing an iterative solution. If this is not a feasible route, then, at the very least, diagram the recursive process undertaken by your code to deepen your knowledge.

- Visualizing Recursion

0 1 1 2 3 5 8 13

$$\text{Fibo}(N) = \text{Fibo}(N-1) + \text{Fibo}(N-2)$$

Recurrence relation When you write Recursion in a formula is called Recurrence relation



package com.Recursion;

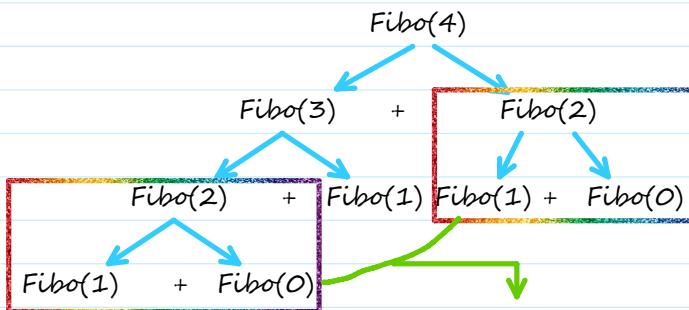
```

public class Fibonacci_num {
    public static void main(String[] args) {
        System.out.println(fibo(50));
    }
    static int fibo(int n){
        //base condition
        if(n<2){
            return n;
        }
        return fibo(n-1)+fibo(n-2);
    }
}
  
```

- How to understand & approach a problem

- 1) Identify if you can break down problem into smaller problem
- 2) Write recurrence relation if needed it.
- 3) Draw the recursive tree
- 4) About the tree

- See the flow of function, how they are getting in stack.
- Identify & focus on left tree calls & right tree calls
- Draw the tree and pointer again and again using pen & paper.
- Use the debugger to see the flow.
- See the how value & what type of value (int, string, etc) are returned at each step. See where the function call will come out. In the end, you will come out of the main function.



Here we can see same thing we are doing again to save time and memory "Dynamic programming" comes into play



Dynamic programming is nothing but recursion with memorization i.e. calculating and storing values that can be later accessed to solve subproblems that occur again, hence making your code faster and reducing the time complexity (computing CPU cycles are reduced).

- Binary Search using Rec

```
package com.Recursion;

public class BinarySearchRec {
    public static void main(String[] args) {
        int[] arr={1,2,3,4,5,66,67,68,69,71,78,79,88};
        int target=2;
        System.out.println(BSR(arr,target,0,arr.length-1));
    }
    static int BSR(int[] arr, int target, int start, int end){
        if(start>end){
            return -1;
        }
        int mid = start+(end-start)/2;
        if(target==arr[mid]){
            return mid;
        }
    }
}
```

```
if(target<arr[mid]){
    return BSR(arr,target,start, mid-1);
}
return BSR(arr,target,mid+1, end);
}
```

# When to use Recursion

02 February 2023 00:24

When should you use recursion?

Recursive code is pretty cool in the sense that you can use recursion to do anything that you can do with non-recursive code. Functional programming is pretty much built around that concept.

But just because you can do something doesn't mean you should do something.

There are a lot of cases where writing a function recursively will be a lot more effort and may run less efficiently than the comparable iterative code. Obviously in those cases we should stick with the iterative version.

So how do you know when to use recursion? Unfortunately, there's no one-size-fits-all answer to this question. But that doesn't mean we can't start to narrow things down.

Here are a couple questions you can ask yourself to decide whether you should solve a given problem recursively:

**Does the problem fit into one of our recursive patterns?**

In the recursive patterns section, we will see 6 different common recursive patterns. One of the easiest ways to decide whether or not to use recursion is simply to consider if the problem fits into one of those patterns.

**Does the problem obviously break down into subproblems?**

Many people define recursion as "solving a problem by breaking it into subproblems". This is a perfectly valid definition, although the 6 recursive patterns get more precise. However, if you see a way to break a problem down into subproblems, then it can likely be solved easily using recursion.

**Could the problem be solved with an arbitrary number of nested for loops?**

Have you ever tried to solve a problem where it would be easy to solve if you could have a number of nested for loops depending on the size of the input? For example, finding all N-digit numbers. We can't do this with actual for loops, but we can do this with recursion. This is a good indicator that you might want to solve a problem recursively.

**Can you reframe the problem as a search problem?**

Depth-first search, one of the patterns we will see, is incredibly flexible. It can be used to solve almost any recursive problem by reframing it as a search problem. If you see a problem that can be solved by searching, then you have a good recursive candidate.

**Is it easier to solve it recursively than iteratively?**

At the end of the day, this is what it comes down to. Is it easier to solve the problem recursively than it is to solve it iteratively? We know that any problem can be solved either recursively or iteratively, so you just have to decide which is easier.

Recurrence formula

★ Base condition

★ Recursive tree

- Print 1 to n

Stack Memory visualization

```
package com.Rec2;
```

```
import java.util.Scanner;
```

```
public class RecIntro {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        print(num);
    }
```

```
static void print(int num) {
    //base condition
    if(num==0){
        return;
    }
    print(num-1);
    System.out.println(num);
}
```

### • Factorial of a number

```
package com.Rec2;
```

```
import java.util.Scanner;
```

```
public class Fact {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        System.out.println(factorial(num));
    }
```

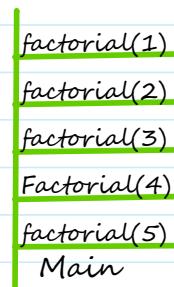
```
static int factorial(int num){
    //base condition
    if(num==1)
        return 1;
    //recurrence formula
    return num * factorial(num-1);
}
```

Suppose num is 05

Num \* f(4) --&gt; 24 \* 5 = 120

↓  
04 \* f(3) --> 4 \* 6↓  
03 \* f(2) --> 3 \* 2↓  
02 \* f(1) --> 2 \* 1

//base condition if f(1) return 1



//stack memory

- Sum of n numbers

```
package com.Rec2;

import java.util.Scanner;

public class Sum {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        System.out.println(sum(num));
    }

    static int sum(int num) {
        //base condition
        if(num==1) return 1;
        return num + sum(num-1);
    }
}
```

- Sum of individually digits

```
package com.Rec2;

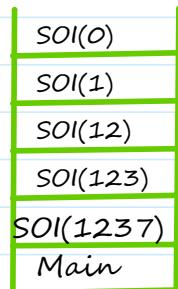
import java.util.Scanner;

public class DigitSum{
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        System.out.println(sumOfIndividalDigits(num));
    }

    static int sumOfIndividalDigits(int num) {
        //base condition
        if(num<=0)
            return 0;
        return (num%10) + sumOfIndividalDigits(num/10);
    }
}
```

Example --> num --> 1237

7 + SOI(123)----->7+6 =13  
3 + SOI(12)----->3+3 =6  
2 + SOI(1) ----->2+1 =3  
1 + SOI(0) ----->1+0 =10



//stack memory

- **Concept**

```
package com.Rec2;

import java.util.Scanner;

public class Concept {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        // System.out.println(fun(num));
        fun(num);
    }

    static void fun(int num) {
        if(num==0) return;
        System.out.println(num);
        fun(--num);
        // fun(num--);
    }
}
```

- `++num` -----> adding 1 then passing num
- `num++` -----> passing num then it will add 1

- **Reverse of a number**

```
package com.Rec2;

import java.util.Scanner;

public class ReverseNo {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        int noOfDigits = (int)(Math.log10(num)+1);
        System.out.println(reverse(num,noOfDigits));
    }

    static int reverse(int num, int noOfDigits) {
        //base condition
    }
}
```

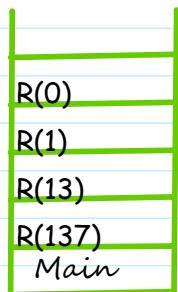
```
if(num%10 == num) return num;  
noOfDigits--;  
return ((num%10)*(int)Math.pow(10,noOfDigits)) + reverse(num/10,noOfDigits);  
  
}  
}
```

Example --> num --> 137

$7*100 + R(13) \rightarrow 700 + 31 = 731$

$3*10 + R(1) \rightarrow 30 + 1 = 31$

$1 + R(0) \rightarrow 1 + 0 = 1$



//stack memory

## Recursion 3

09 June 2022 15:44

- Check whether the array is sorted or not ?

```
package com.Rec3;

import java.util.Scanner;

public class SortedArr {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = in.nextInt();
        }
        System.out.println(isItSorted(arr, 0));
    }

    static boolean isItSorted(int[] arr, int index) {
        //base condition
        if(index == arr.length-1) return true;

        return (arr[index] < arr[index+1]) && isItSorted(arr, index+1);
    }
}
```

- Linear Search using Recursion

```
package com.Rec3;

import java.util.ArrayList;
import java.util.Scanner;

public class LinearSearch {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int[] arr = new int[5]; //it will create 5 empty spaces not 6.
        for (int i = 0; i < arr.length; i++) {
            System.out.println("value for index "+i);
            arr[i] = in.nextInt();
        }
        System.out.println("enter the value you want to search");
        int value = in.nextInt();

        search(arr, value, 0);
        System.out.println(list);

        // System.out.println(search2(arr, value, 0, new ArrayList<>()));
    }
}
```

```
}
```

```
/*
```

1- we have to return the index whether it is there or not

2- if the targeted value is there more than one then return that index also  
so to deal this problem we will use array list concept  
to print array list there is a numerous way lets see --->  
\*/

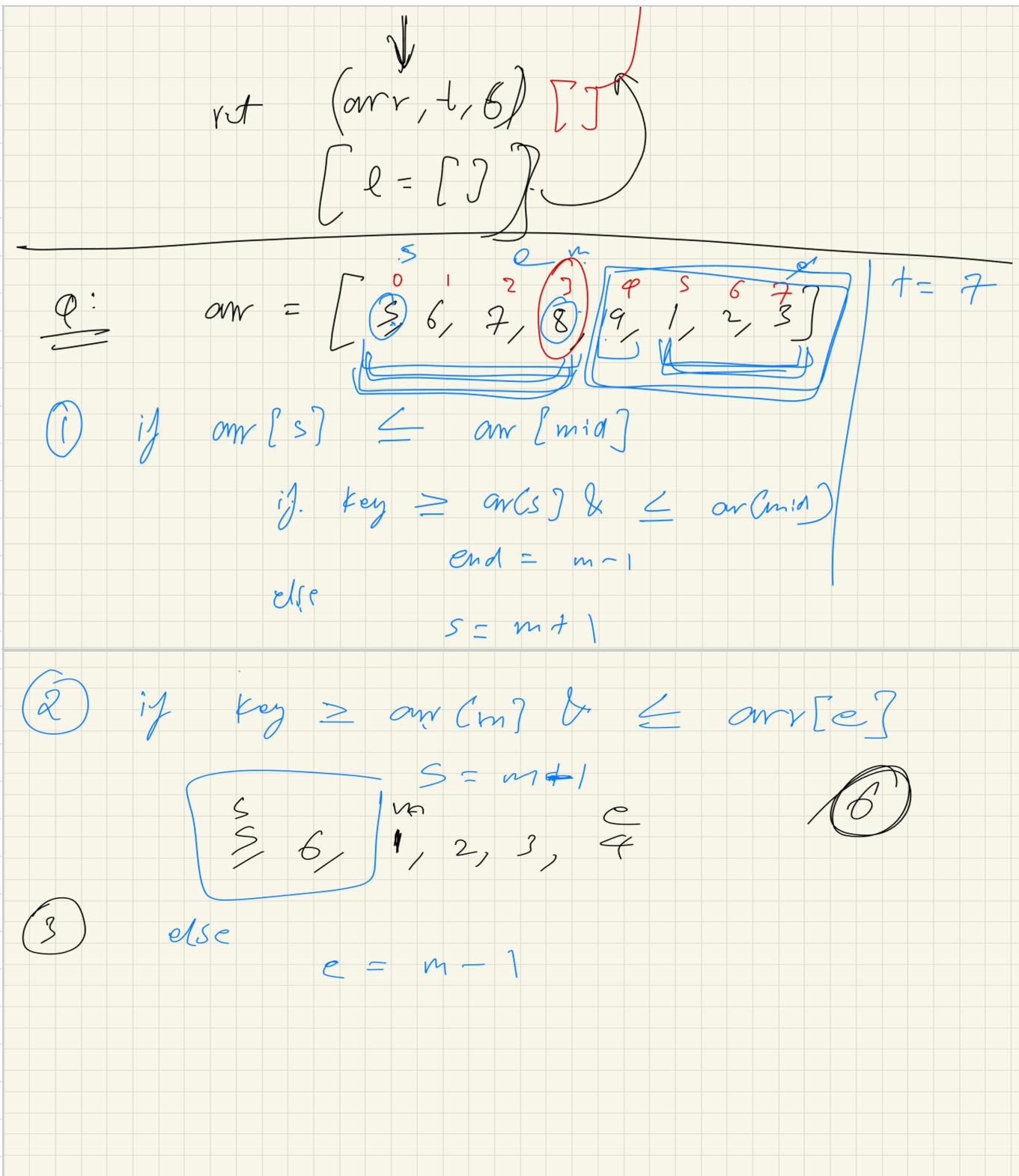
//1st way ->

```
static ArrayList<Integer> list = new ArrayList<>();  
static void search(int[] arr, int value, int index) {  
    if(arr[index]==value) {  
        list.add(index);  
    }  
    //base condition  
    if(index == arr.length-1) {  
        return;  
    }  
    search(arr,value,index+1);  
}
```

//2nd way->

```
static ArrayList<Integer> search2(int[] arr, int value, int index, ArrayList<Integer> list1) {  
    if(arr[index]==value) {  
        list1.add(index);  
    }  
    //base condition  
    if(index == arr.length-1) {  
        return list1;  
    }  
    return search2(arr,value,index+1,list1);  
}
```

- **Rotated Binary Search using Recursion**



//program

```

package com.Rec3;

public class Rotated_BS {
  public static void main(String[] args) {
    int[] arr = {5, 6, 7, 8, 9, 1, 2, 3};
    System.out.println(search(arr, 4, 0, arr.length - 1));
  }
}
  
```

```
static int search(int[] arr, int target, int s, int e) {  
    if (s > e) {  
        return -1;  
    }  
  
    int m = s + (e-s) / 2;  
    if (arr[m] == target) {  
        return m;  
    }  
  
    if (arr[s] <= arr[m]) {  
        if (target >= arr[s] && target <= arr[m]) {  
            return search(arr, target, s, m-1);  
        } else {  
            return search(arr, target, m+1, e);  
        }  
    }  
  
    if (target >= arr[m] && target <= arr[e]) {  
        return search(arr, target, m+1, e);  
    }  
  
    return search(arr, target, s, m-1);  
}  
}
```

## Recursion 4

11 June 2022 03:11

- Print pattern using Recursion

```
package com.Rec4;

public class Pat1 {
    public static void main(String[] args) {
        pattern(4,0);
        pattern2(4,0);
    }
    static void pattern(int row, int col){
        //base condition
        if(row==0) return;

        if(col<row){
            System.out.print("*");
            pattern(row,col+1);
        }else {
            System.out.println();
            pattern(row-1, 0);
        }
    }
    static void pattern2(int row, int col){
        //base condition
        if(row==0) return;

        if(col<row){
            pattern2(row,col+1);
            System.out.print("*");
        }else {
            pattern2(row-1, 0);
            System.out.println();
        }
    }
}
```

Pattern 1  
\*\*\*\*  
\*\*\*  
\*\*  
\*

Pattern 2  
\*  
\*\*  
\*\*\*  
\*\*\*\*

- Do bubble sort using Recursion

```
package com.Rec4;

import java.util.Arrays;
import java.util.Stack;

public class BubbleSort {
    public static void main(String[] args) {
        int[] arr={2,3,9,5,4,44,42,41};
        bS(arr, arr.length);
        System.out.println(Arrays.toString(arr));
    }

    static void bS(int arr[], int n)
    {
        // Base condition
        if (n == 1)
            return;

        // One pass of bubble sort. After
        // this pass, the largest element
        // is moved (or bubbled) to end.
        for (int i=0; i<n-1; i++)
            if (arr[i] > arr[i+1])
            {
                // swap arr[i], arr[i+1]
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
    }
}
```

```
// Largest element is fixed,  
// recur for remaining array  
bS(arr, n-1);  
}  
}
```

- Selection sort using Recursion

```
package com.Rec4;  
  
import java.util.Arrays;  
  
public class SelectionSort {  
    public static void main(String[] args) {  
        int[] arr={1,3,2,4,7,5,6};  
        sS(arr, arr.length-1);  
        System.out.println(Arrays.toString(arr));  
    }  
    static void sS(int[] arr, int k){  
        int max=arr[0];  
        boolean flag=false;  
        int maxI=0;  
        //base condition  
        if(k==0) return;  
        for (int i = 0; i <=k; i++) {  
            if(arr[i]>max){  
                max=arr[i];  
                flag=true;  
                maxI=i;  
            }  
        }  
        if(flag){  
            swap(arr,maxI,k);  
        }  
        sS(arr,k-1);  
    }  
    static void swap(int[] arr,int index1, int index2){  
        int temp=arr[index1];  
        arr[index1]=arr[index2];  
        arr[index2]=temp;  
    }  
}
```

# Time complexity and space complexity

10 May 2022 01:03

# Operators

10 May 2022 16:38

Operators are the symbols used to perform specific operations. There are various operators that can be used for different purposes.

The operators are categorized as:

Unary

Arithmetic

Relational

Logical

Ternary

Assignment

Bitwise

**Unary operators** act upon only one operand and perform operations such as increment, decrement, negating an expression or inverting a boolean value.

Operator	Name	Description
++	post increment	increments the value after use
	pre increment	increments the value before use
--	post decrement	decrements the value after use
	pre decrement	decrements the value before use
~	bitwise complement	flips bits of the value
!	logical negation	Inverts the value of a boolean

E.g.:

```
public static void main(String args[]){  
    int numOne = 10;  
    int numTwo = 5;  
    boolean isTrue = true;  
    System.out.println(numOne++ + " " + ++numOne); //Output will be 10 12  
    System.out.println(numTwo-- + " " + --numTwo); //Output will be 5 3  
    System.out.println(isTrue + " " + ~numOne); //Output will be false -13  
}
```



bitwise operator changes 0 to 1 and 1 to 0 and bitwise complement of integer N is equal to  $-(N+1)$   
here  $-(12+1) = -13$

**Arithmetic operators** are used to perform basic mathematical operations like addition, subtraction, multiplication and division.

Operator	Description
+	additive operator (also used for String concatenation)
-	subtractive operator
*	multiplication operator
/	division operator
%	modulus operator

E.g.:

```
public static void main(String args[]) {
    int numOne = 10;
    int numTwo = 5;
    System.out.println(numOne + numTwo); //Output will be 15
    System.out.println(numOne - numTwo); //Output will be 5
    System.out.println(numOne * numTwo); //Output will be 50
    System.out.println(numOne / numTwo); //Output will be 2
    System.out.println(numOne % numTwo); //Output will be 0
}
```

**Relational operators** are used to compare two values. The result of all the relational operations is either true or false.

Operator	Description
==	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
!=	not equal to

E.g.:

```
public static void main(String args[]) {
    int numOne = 10;
    int numTwo = 5;
    System.out.println(numOne > numTwo); //Output will be true
}
```

**Logical operators** are used to combine two or more relational expressions or to negate the result of a relational expression.

Operator	Name	Description
&&	AND	result will be true only if both the expressions are true
	OR	result will be true if any one of the expressions is true
!	NOT	result will be false if the expression is true and vice versa

Assume A and B to be two relational expressions. The below tables show the result for various logical operators based on the value of expressions, A and B.

A	B	A&&B	A B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

A	!A
true	false
false	true

E.g.:

```
public static void main(String args[]) {
    int numOne = 100;
    int numTwo = 20;
    int numThree = 30;
    System.out.println(numOne > numTwo && numOne > numThree); //Output will be true
}
```

**Ternary operator** is used as a single line replacement for if-then-else statements and acts upon three operands.

Syntax:

<condition> ? <value if condition is true> : < value if condition is false>

E.g.:

```
public static void main(String args[]) {
    int numOne = 10;
    int numTwo = 5;
    int min = (numOne < numTwo) ? numOne : numTwo;
    System.out.println(min); //Output will be 5
}
```

**Assignment operator** is used to assign the value on the right hand side to the variable on the left hand side of the operator.

Some of the assignment operators are given below:

Operator	Description
=	assigns the value on the right to the variable on the left
+=	adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left
-=	subtracts the value of the variable on the right from the current value of the variable on left and then assigns the result to the variable on the left
*=	multiples the current value of the variable on left to the value on the right and then assigns the result to the variable on the left
/=	divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left

E.g.:

```
public static void main(String args[]) {
    int numOne = 10; //The value 10 is assigned to numOne
    System.out.println(numOne); //Output will be 10
    numOne += 5;
    System.out.println(numOne); //Output will be 15
    numOne -= 5;
    System.out.println(numOne); //Output will be 10
    numOne *= 5;
    System.out.println(numOne); //Output will be 50
    numOne /= 5;
    System.out.println(numOne); //Output will be 10
}
```

**Bitwise operators** are used to perform manipulation of individual bits of a number.

Before we take a look at the different bitwise operators, let us understand how to convert a decimal number to

binary number and vice versa.

The decimal or the base 10 number system is used in everyday life but the binary number system is the basis for representing data in computing systems.

You will now see how to convert a decimal number to binary number.

Step 1:

Divide the decimal number by 2.

Step 2:

Write the number on the right hand side. This will be either 1 or 0.

Step 3:

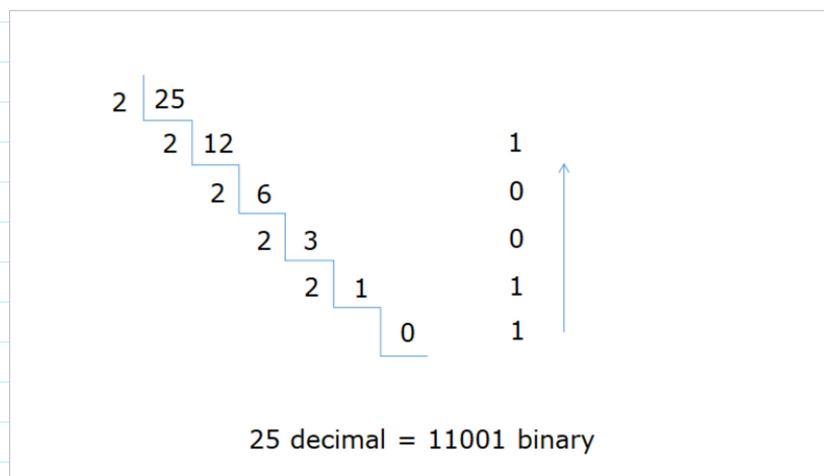
Divide the result of the division and again by 2 and write the remainder.

Step 4:

Continue this process until the result of the division is 0.

Step 5:

The first remainder that you received is the least significant bit and the last remainder is the most significant bit.



You will now see how to convert the binary number back to decimal number.

The decimal number is equal to the sum of binary digits ( $d_n$ ) times their power of 2 ( $2^n$ ).

Lets take the example of 11001.

$$11001 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 0 + 0 + 1 = 25$$

$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
1	1	0	0	1

$$16 + 8 + 0 + 0 + 1 = 25$$

## Operator precedence

Operators also have precedence. The below table lists the operators according to the precedence from highest to lowest.

Operators with higher precedence are evaluated before the operators with lower precedence. When an expression has two operators with the same precedence one after the other, the expression is evaluated according to their associativity. The associativity of operators can be left-to-right, right-to-left or no associativity.

For example: num1 = num2 = num3 = 39 is treated as (num1 = (num2 = (num3 = 39))), leaving all three variables with the value 39, since the = operator has right-to-left associativity. On the other hand, 84 / 4 / 3 is treated as ((84 / 4) / 3) since the / operator has left-to-right associativity. Some operators, like the postfix and prefix operators, are not associative: for example, the expression num++-- is invalid.

Operator Name	Operator
parenthesis/ brackets	()
postfix	++, --
unary, prefix	++, --, +, -, ~, !
multiplicative	*, /, %
additive	+, -
shift	<<, >>, >>>
relational	<, >, <=, >=
equality	==, !=
logical AND	&&
logical OR	
ternary	?:
assignment	=, +=, -=, *=, <<=, >>=, >>>=

# Number System

28 May 2022 11:24

# Conversion

28 May 2022 11:25

✓ •  $>>$  (right shift)

Suppose  $a = 10 >> 2$   
 $a = 10 >> 3$

First it will convert 10 into binary  
 Then it will do right shifts according to instruction

Example :- 10 --> 1010 --> 0010 --> 2  
 :- 10 --> 1010 --> 0001 --> 1

Tip :  $10 >> 2$   
 $\downarrow$   
 $10/2=5$   
 $5/2=2$

$a \ll 1 = 2a$
General point --> $a \ll b = a * 2^b$

✓ •  $<<$  (left shift)

Suppose  $a = 10 << 2$   
 $a = 10 << 3$

First it will convert 10 into binary  
 Then it will do left shifts according to instruction

Example :- 10 --> 00001010 --> 00101000 --> 40  
 :- 10 --> 00001010 --> 01010000 --> 80

Tip :  $10 << 2$   
 $\downarrow$   
 $10 * 2 = 20$   
 $20 * 2 = 40$

$a \ll 1 = a/2$
General point --> $a \ll b = a/2^b$

Question 1: Given a number n find if it is odd or even

Point : Every number is calculated in binary form internally

Example -  $(19)_{10} = (10011)_2$

$$19 \rightarrow 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$

1	0	0	1
---	---	---	---



This will be even. Bcz  
 the power of 2 always  
 be even only

1
---

So, the number is  
 dependent on last no.  
 weather the no. is even  
 or odd.  
 Leaving this, every  
 other it is power of 2

Hence : if the last digit of binary is 1 than its odd.  
 if the last digit of binary no. is 0 than its even.

100101  
000001 &  
-----  
000001 -----> (1) hence, odd

Sum up:  $n \& 1 == 1 \Rightarrow$  odd else even

```
//program
package com.Math;

import java.util.Scanner;

public class Even_Odd {
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int num=in.nextInt();
        System.out.println(isOdd(num));
    }

    static boolean isOdd(int num) {
        return (num&1)==1;
    }
}
```

Question 2: every no has duplicate of that except one. Return that number

arr = [2, 3, 4, 1, 2, 1, 3, 6, 4]

As we know xor of same element is zero

05 June 2022 15:08

# Maths for dsa

05 June 2022 15:52

- Whether the no is prime or not?

```
//program
package com.Math2;

import java.util.Scanner;

public class IsPrime {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        System.out.println(isPrime(num));
    }

    static boolean isPrime(int num) {
        int newNum;
        boolean flag;

        //perfect sqrt of a give num
        int a = (int)(Math.sqrt(num)*Math.sqrt(num));
        int b = (int)((Math.sqrt(num)+1)*(Math.sqrt(num)+1));

        if((num-a)<(b-num)){
            newNum=a;
        }else
            newNum=b;

        for (int i = 2; i <=Math.sqrt(newNum); i++) {
            boolean prime=true;

            // whether the i is prime or not -->
            for (int j = 2; j <i/2 ; j++) {
                if(i%j==0)
                    prime=false;
            }
            //check whether the no. is divisible by prime no. or not
            //if yes return false
            //if no return true
            if(prime && num%i==0)
                return false;
        }
        return true;
    }
}
```

- Range is given --- find which which no. is prime ?

```

//program
package com.Math2;

import java.util.Scanner;

public class Prime_range {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int range1= in.nextInt();
        int range2 = in.nextInt();
        for (int i = range1; i <=range2; i++) {
            if(i == 0){
                System.out.println(i+" is neither prime nor composite");
                continue;
            }
            System.out.println(i+" "+ isprime(i));
        }
    }

    static boolean isprime(int num) {

        if(num == 1) return false;

        int newNum;
        int sr =(int)Math.sqrt(num);
        int a =sr*sr;
        int b = (sr+1)*(sr+1);

        if((num-a)<(b-num))
            newNum=a;
        else newNum=b;

        for (int i = 2; i <=Math.sqrt(newNum) ; i++) {
            boolean prime=true;
            for (int j = 2; j < i/2; j++) {
                if(i%j==0) prime=false;
            }
            if(prime && num%i==0) return false;
        }
        return true;
    }
}

```

- **Find the SquareRoot Of the no.**

```

//program
package com.Math2;

import java.util.Scanner;

```

```

public class SQRT {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num=in.nextInt();
        System.out.println("The sqrt of a given num is "+sqrt(num));
    }

    static int sqrt(int num) {
        int sr = (int)Math.sqrt(num);
        int a = sr*sr;
        int b = (sr+1)*(sr+1);
        if((num-a)<(b-num))
            return (int)Math.sqrt(a);
        else return (int)Math.sqrt(b);
    }
}

```

- Find the factors of no.

Suppose the num is : 20

$$\begin{aligned}
 1 * 20 &= 20 \\
 2 * 10 &= 20 \\
 4 * 5 &= 20 \\
 \\ 
 5 * 4 &= 20 \\
 10 * 2 &= 20 \\
 20 * 1 &= 20
 \end{aligned}$$

//so optimized way to print the answer is  
 ---> Print i & Print num/i  
 ---> No we don't need to run loop till num  
 ---> we have to run now, till  $\sqrt{\text{num}}$

```

//program
package com.Math2;

import java.util.ArrayList;
import java.util.Scanner;

public class Factor {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num=in.nextInt();
        factor(num);
        factor2(num);
    }

    static void factor(int num){
        for (int i = 1; i <=Math.sqrt(num) ; i++) {
            if(num%i==0){
                if(num/i==i) System.out.println(i);
                else

```

1 20 2 10 4 5

```

        if(num%i==0){
            if(num/i==i) System.out.println(i);
            else{
                System.out.print(i+" ");
                System.out.print(num/i+" ");
            }
        }
    }
}

```

1 2 0 2 10 4 5

```

//printed the answer in sorted way :
static void factor2(int num){
    ArrayList<Integer> list = new ArrayList<>();
    for (int i = 1; i <=Math.sqrt(num) ; i++) {
        if(num%i==0){
            if(num/i==i) System.out.println(i);
            else{
                System.out.print(i+" ");
                list.add(num/i);
            }
        }
    }
    for (int i = list.size()-1; i>=0; i--) {
        System.out.print(list.get(i)+" ");
    }
}

```

1 2 4 5 10 20

- HCF

$$\text{Hcf}(4, 18) = 2$$

Factors of 4 = 1, 2, 4

Factors of 18 = 1, 2, 3, 6, 9

--> 2 is the highest common factor of 4 and 18

a, b

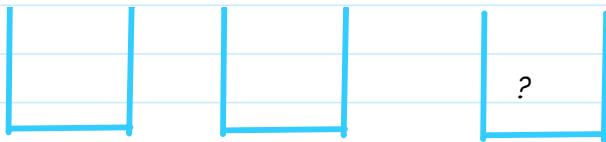
$ax + by = L$  //we have to tell whether we will be able to full the given bucket (no of liters will be mentioned how much lt we need to full) from the 2 buckets

2 lt

4 lt

5 lt

?



Note to remember

- It's have to be exactly 5L.
- No scale is there in bucket that you will measure and put accordingly.

Let's see how?

$$2x + 4y = 5$$

$$2(x+2y) = 5 \rightarrow 2 \text{ we got by taking hcf}(2, 4) = 2$$

$$x + 2y = 2.5$$

//The answer is no: we can't fill the third bucket from these 2 buckets

Example 2

$$3x + 6y = 9$$

$$3(x+2y) = 9$$

$$x + 2y = 3$$

//The answer is yes. Now we can fill the third bucket from these 2 bucket

Euclid's Algorithm :

$$\gcd(a, b) = \gcd(\text{rem}(b, a), a)$$

$$\begin{aligned} \gcd(105, 224) &= \gcd(\text{rem}(224, 105), 105) \\ &= \gcd(14, 105) \end{aligned}$$

Why?

$$105x + 224y$$

$\downarrow$  why subtract?

$$14n + 105y$$

i.e.

bewere the gcd of  $(105, 214)$

also divides a linear combination  
of  $105$  &  $214$ .

$$\text{Ex: } \underline{\underline{214 - 2 \times 105}} = 14 \text{ (rem)}$$

Lcm:

$\text{lcm}(a, b) = \min$ .  
no. divisible  
by both ab

$$\text{lcm}(2, 4) = 4$$

$$(3, 7) = 21$$

Note:

Say we have  $a, b$

$$d = \gcd(a, b)$$

$$f = \frac{a}{d}, \quad g = \frac{b}{d}$$

$$\Rightarrow a = fd, \quad b = gd$$

$\text{lcm} = c$        $\cancel{\text{lcm}}(a, b) = \text{lcm}(fd, gd)$

$\cancel{*}$  We know that  $f$  &  $g$  will have no other common factors.

$$a = 9, \quad b = 18$$

$$d = 9$$

$$f = 1, \quad g = 2$$

Say,  $hg = 3 \times 3 = 9$  (wrong)  
 ↓ bigger       $f = \frac{9}{3}, 3$        $g = \frac{18}{3}, 6$

$$\cancel{*} \quad a = fd \quad b = gd$$

$$\text{lcm} = f \times g \times d$$

This is how  
above conditions  
are satisfied.

$$\underline{\text{More info:}} = a \times b$$

$$= f \cdot d \times g \times d \rightarrow d \text{ is repwdly, hence remain}$$

$$17, 19$$

$$l_{am} = f \times g \times d$$

$$\begin{aligned} a \times b &= fd + gd \\ &= d \times fg \\ &= h_{ij} \times l_{am} \\ h_{ij} \times l_{am} &= a \times b \end{aligned}$$

formula!

$$L_{im}(a,b) = \frac{a \times b}{HC F(a,b)}$$

# Maths2

06 June 2022 02:31



Maths2





Prime Nos: 2, 3, 5, 7, 13, ...

~~2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13~~

```
for( i=2; i < N; i++ ) {  
    if ( N % i ) {  
        Not Prime;  
    }  
}
```

*Prime*

Another Example:

1	X	36
2	X	18
3	X	12
4	X	9
6	X	6
9	X	4
12	X	3
18	X	2
36	X	1

This is rejected.  
hence ignore.

3 ≠ 12

12 ≠ 3

Hence, only  
marks circles  
for numbers  $\leq$   
 $\sqrt{n}$

$$C \leq \text{Sqrt}(N)$$

$$C \times C \leq N$$

Q:  $N = 40$

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37

2	3	X	5	X	7	X	X	X
11	X	13	X	X	17	X	19	X
X	X	23	X	X	X	X	29	X
31	X	X	X	X	37	X	X	X

$O \rightarrow F$   
 $X \rightarrow T$

Time Complexity:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \dots$$

$$n \left( \frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \dots \right)$$

Harmomic progression for primes.

$$\log(\log n)$$

Total time complexity:  $O(N + \log(\log n))$

## Finding square root of a number

10

18

36

$$\text{if } (m \times m > n) \\ e = m - 1$$

else

$$s = m + 1$$

$$\text{Sqr}t(40) = 6.\underline{\hspace{2mm}32} \\ \text{above way} \quad ?$$

$$\begin{aligned} \text{root} &= 6.1 \\ &= 6.2 \\ &= 6.3 \\ &= 6.4 \end{aligned}$$

0.1

same thing for 0.81

## Newton Raphson method

$$\text{root} = \underbrace{\left( X + \frac{N}{X} \right)}_2$$

And by root  
 $= \sqrt{N}$   
 sqrt you have assumed

$$\text{error} = |\text{root} - X|$$

You will find your  
ans when error < 1

- ① Assign  $X$  to  $N$
- ②  $\xleftarrow{\quad}$
- ③ Update the value of  $X = \text{root}$

Complexity:  $O((\log N) f(n))$

$f(n)$  = work of calculating  $\frac{f(n)}{f'(n)}$   
with  $n$ -digit precision.

Why the formula works?

$$\sqrt{N} = \underbrace{\left( X + \frac{N}{X} \right)}_2$$

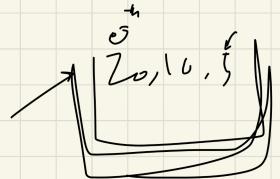
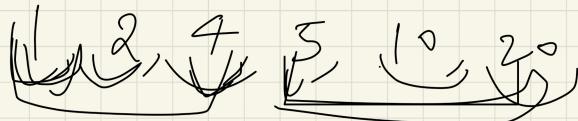
$$\sqrt{N} = \underbrace{\sqrt{N} + \frac{\sqrt{N}}{\overline{\sqrt{N}}}}_{\text{ }} \Rightarrow \sqrt{N} = \underbrace{\frac{2\sqrt{N}}{2}}$$

$$\sqrt{N} = \frac{\sqrt{N} + \frac{\sqrt{N}}{\sqrt{N}}}{2} \Rightarrow \sqrt{N} = \frac{2\sqrt{N}}{2}$$

$$\sqrt{N} = \boxed{\sqrt{N}}$$

Factors of a number:

$$n = 20 \Rightarrow$$



$$20 \% 1 \checkmark \Rightarrow 20 * 1 = 20$$

$$20 \% 2 \checkmark \Rightarrow 10 * 2 = 20$$

$$20 \% 4 \checkmark \Rightarrow 5 * 4 = 20$$

$$20 \% 5 \checkmark = 4 * 5 = 20$$

$$20 \% 10 = 2 * 10 = 20$$

*repeated*

## Properties of modulo (%)

$$\star (a+b)\%m = ((a \% m) + (b \% m)) \% m$$

$$\star (a-b)\%m = ((a \% m) - (b \% m) + m) \% m$$

$$\star (a * b) \% m = ((a \% m) * (b \% m)) \% m$$

$$\star \left(\frac{a}{b}\right) \% m = ((a \% m) * (b^{-1} \% m)) \% m$$

$b^{-1} \% m$   $\neq$  Multiplicative modulo inverse ( $m \neq 1$ )

Ex:  $(6 * y) \% 7 = 1$

$$y = m \mid \text{ for } 6 \quad \& \quad y = 6$$

$$(6 * 6) \% 7 = 36 \% 7 = 1$$

$m \mid = b^{-1} \% m$  means that

$b \& m \& \text{ co-prime.}$

$$\star (a \% m) \% n = a \% m$$

$$\star m \% m = 0 \quad \forall n \in \text{the integers.}$$

Extra: If  $p$  is prime no. which is not a divisor of  $d$ , then  $ab^{p-1} \not\equiv p \pmod{p}$  due to Fermat's Little theorem.

How? will be covered in  
advance DJ course :)

### Die-hard Example:

$$\begin{array}{c|c|c} 3 & 5 & = \\ \hline a & b & 4 \end{array}$$

$$\begin{aligned} 1^{\text{st}} \rightarrow & (0, 0) \rightarrow (3, 0) \rightarrow (0, 3) \\ 2^{\text{nd}} \rightarrow & (0, 3) \rightarrow (3, 3) \rightarrow (1, 5) \\ & (0, 1) \leftarrow (1, 0) \\ 3^{\text{rd}} \rightarrow & (0, 1) \rightarrow (3, 1) \rightarrow (0, 4) \end{aligned}$$

Intuition:  
 If we do  $a \rightarrow 2^1$  times  
 and  $b \rightarrow 2^2$  times

Ans!

$$\left[ \begin{array}{l} r = as' - bs^2 \\ r = as' + (-bs^2) \end{array} \right] \quad \left[ \begin{array}{l} L = s'a + t'b \\ s'a = L - t'b \end{array} \right]$$

$$r = s'a + t'b - t'b - bs^2$$

$$r = L - (t' + u)b$$

If  $t' + u \neq 0 \Rightarrow [r < 0 \text{ or } r > b]$   
which is not true

$$t' + u = 0 \Rightarrow u = -t'$$

$$r = s'a + t'b = L$$

$\downarrow$

a.k.a  $\rightarrow$   $r = ax + by$

$$3x + 5y = 4$$

?

Put  $x$  &  $y$  as integers, what is the minimum free value you can have of  $cst.$

$$x = -3, \quad y = 2$$

$$3x + 5y = 1$$

minimizing tree values but  
1 can form

~~This is called HCF:~~

HCF of  $a$  &  $b$  = min. pos. value  
of  $\epsilon^m$   $(a + b)$   
where  $a$  &  $b$  are ints.

$$\text{HCF}(4, 18) = 2$$

1, 2, 4

1, 3, 6, 9, 18

*Ans*

$$\text{HCF}(3, 9) = 3$$

1, 3

1, 3, 9

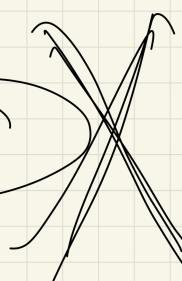
$$\begin{aligned} \min(3x + 9y) &= 3 \\ 3x + 9y \\ 3(x + 3y) \\ = 3(-2 + 3) &= 3 \end{aligned}$$

a, b

$$an + by = L$$

$$2x + 4y = 5$$

$$2(x + 2y) = 5$$
$$x + 2y = 2.5$$



Note:  
What one  $x$ ,  
you will get,  
 $x$  will  
come out  
as a common.

Please wait while OneNote loads this Printout...



×

## Euklid's Algorithm:

$$\gcd(a, b) = \gcd(\text{rem}(b, a), a)$$

$$\begin{aligned} \gcd(105, 224) &= \gcd(\text{rem}(224, 105), 105) \\ &= \gcd(14, 105) \end{aligned}$$

why?

$$105x + 224y$$



why subtract?

$$14x + 105y$$

i.c

because the gcd of  $(105, 224)$   
also divides a linear combination  
of  $105$  &  $224$ .

Ex:  $224 - 2 \cdot 105 = 14$  (rem)

## Lcm:

$\text{lcm}(a, b) = \text{No. divisible}^{\min}$   
by both a & b

$$\text{lcm}(2, 4) = 4$$

$$(3, 7) = 21$$

Note:-

Say we have a, b

$$d = \text{gcd}(a, b)$$

$$f = \frac{a}{d}, \quad g = \frac{b}{d}$$

$$\Rightarrow a = fd, \quad b = gd$$

$\text{lcm} = c$   ~~$\text{lcm}(a, b) = \text{lcm}(fd, gd)$~~

\* We know that f & g will have no other  
common factor.

$$a = 9, \quad b = 18$$

$f = 1$  ,  $(g) = 2$

$$(d = 9)$$

Say,  $\begin{matrix} h \\ \downarrow \text{bifur} \end{matrix} = 3 * 3 = 9$  ✓

$$f = \frac{9}{3} \quad g = \frac{18}{6}$$

(why?)

$\cancel{\star} \quad a = f d \quad b = g d$

$\text{lcm} = f * g * d$

This is how  
above conditions  
are satisfied.

more info:  $= a * b$

$$= f * d * g * d$$

17, 19

$\rightarrow d$  is repeating, hence remain

$\text{lcm} = f * g * d$

$\cancel{\star} \quad a * b = f d * g d$   
 $= d * d f g$

$$= h \cancel{d} * \text{lcm}$$

$$h \cancel{d} * \text{lcm} = a * b$$

formula!

$$\text{Lcm}(a, b) = \frac{a \times b}{\text{HCF}(a, b)}$$

Please wait while OneNote loads this Printout...



x

24 September 2022 13:38

# Merge Sort code

13 June 2022 17:48

```
package com.kunal.sorting;

import java.util.Arrays;

public class MergeSort {
    public static void main(String[] args) {
        int[] arr = {5, 4, 3, 2, 1};
        mergeSortInPlace(arr, 0, arr.length);
        System.out.println(Arrays.toString(arr));
    }

    static int[] mergeSort(int[] arr) {
        if (arr.length == 1) {
            return arr;
        }

        int mid = arr.length / 2;

        int[] left = mergeSort((Arrays.copyOfRange(arr, 0, mid)));
        int[] right = mergeSort((Arrays.copyOfRange(arr, mid, arr.length)));

        return merge(left, right);
    }

    private static int[] merge(int[] first, int[] second) {
        int[] mix = new int[first.length + second.length];

        int i = 0;
        int j = 0;
        int k = 0;

        while (i < first.length && j < second.length) {
            if (first[i] < second[j]) {
                mix[k] = first[i];
                i++;
            }
            else {
                mix[k] = second[j];
                j++;
            }
            k++;
        }

        if (i < first.length) {
            for (int l = i; l < first.length; l++) {
                mix[k] = first[l];
                k++;
            }
        }
        else {
            for (int l = j; l < second.length; l++) {
                mix[k] = second[l];
                k++;
            }
        }

        return mix;
    }
}
```

```
        } else {
            mix[k] = second[j];
            j++;
        }
        k++;
    }

    // it may be possible that one of the arrays is not complete
    // copy the remaining elements
    while (i < first.length) {
        mix[k] = first[i];
        i++;
        k++;
    }

    while (j < second.length) {
        mix[k] = second[j];
        j++;
        k++;
    }

    return mix;
}

static void mergeSortInPlace(int[] arr, int s, int e) {
    if (e - s == 1) {
        return;
    }

    int mid = (s + e) / 2;

    mergeSortInPlace(arr, s, mid);
    mergeSortInPlace(arr, mid, e);

    mergeInPlace(arr, s, mid, e);
}
```

```
private static void mergeInPlace(int[] arr, int s, int m, int e) {  
    int[] mix = new int[e - s];  
  
    int i = s;  
    int j = m;  
    int k = 0;  
  
    while (i < m && j < e) {  
        if (arr[i] < arr[j]) {  
            mix[k] = arr[i];  
            i++;  
        } else {  
            mix[k] = arr[j];  
            j++;  
        }  
        k++;  
    }  
  
    // it may be possible that one of the arrays is not complete  
    // copy the remaining elements  
    while (i < m) {  
        mix[k] = arr[i];  
        i++;  
        k++;  
    }  
  
    while (j < e) {  
        mix[k] = arr[j];  
        j++;  
        k++;  
    }  
  
    for (int l = 0; l < mix.length; l++) {  
        arr[s+l] = mix[l];  
    }  
}
```

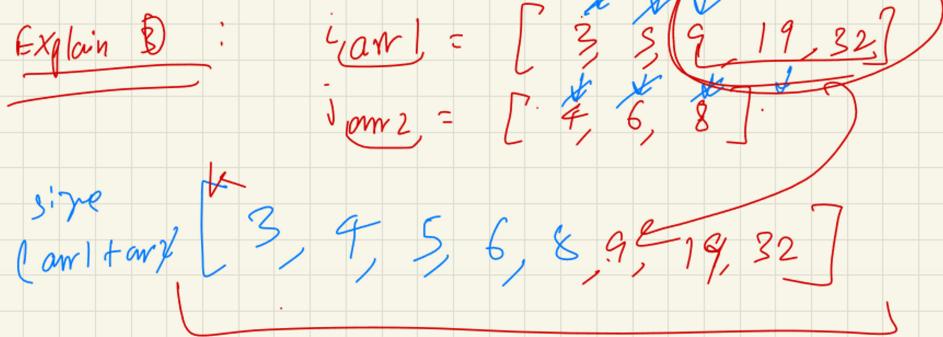


# Merge sort note

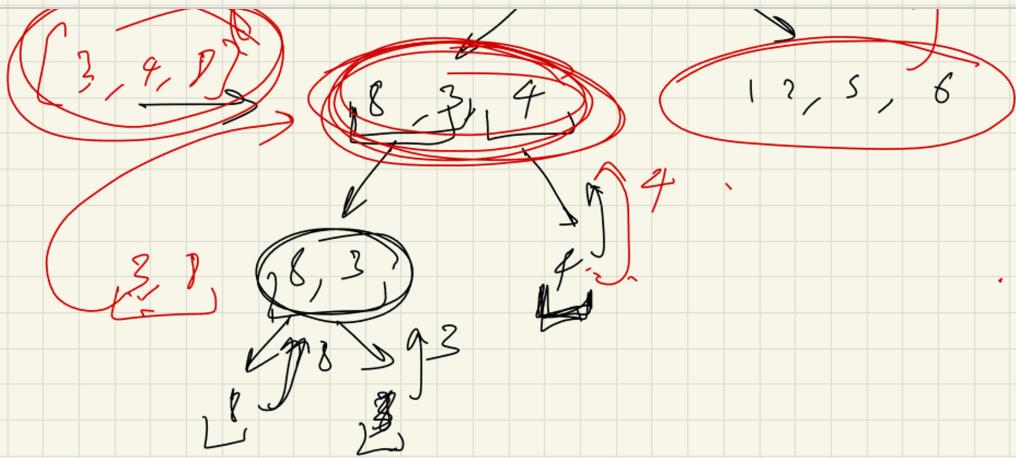
14 June 2022 02:34

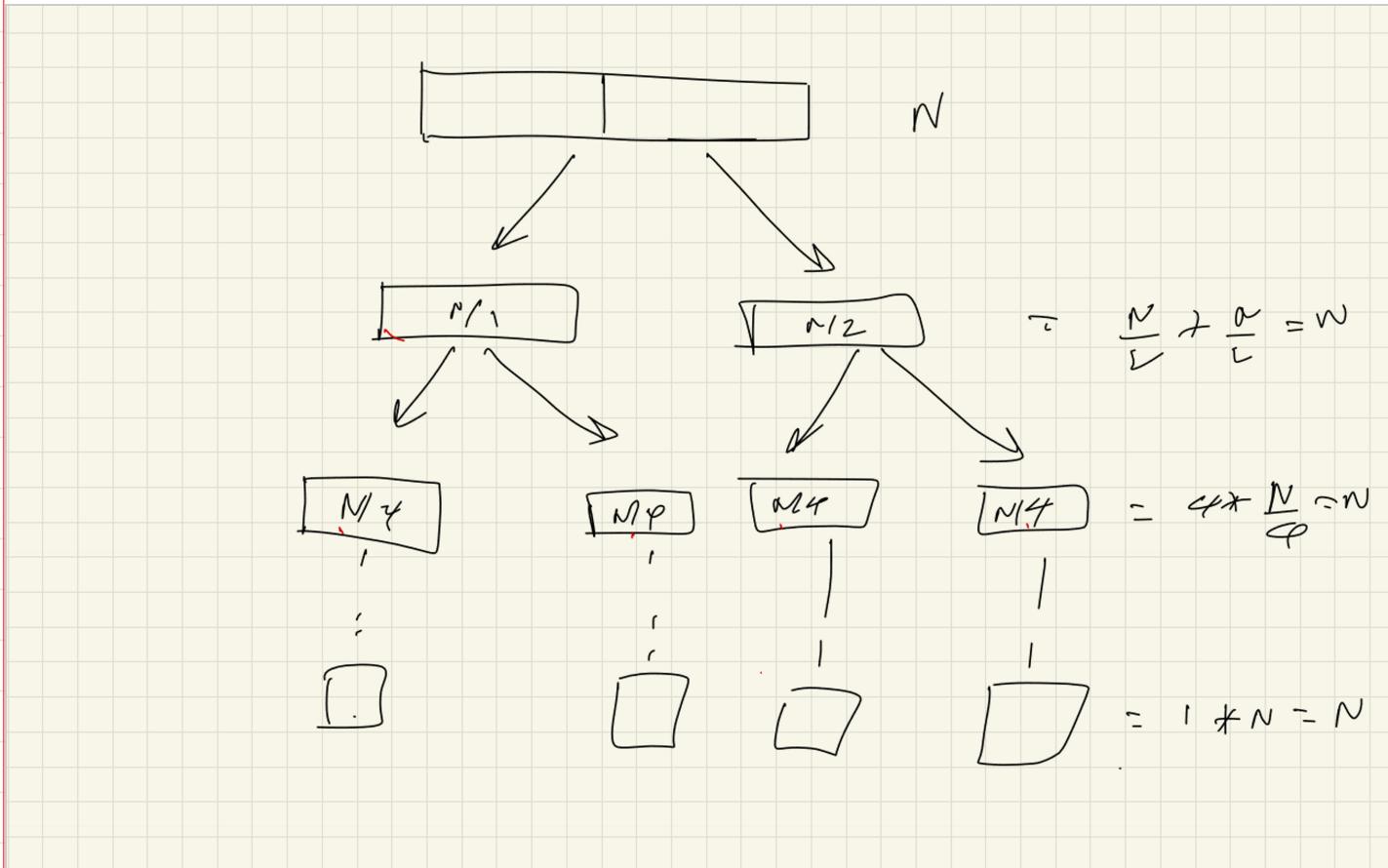
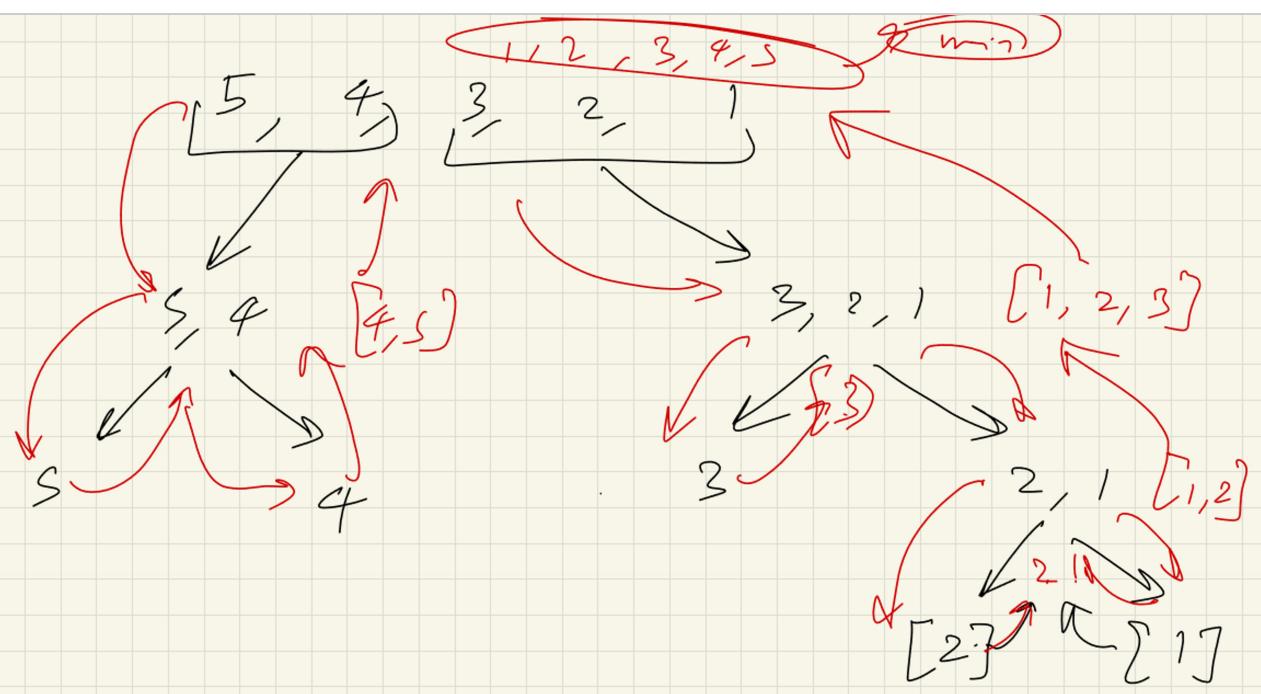
- ① Divide array into 2 parts.
- ② Get both parts sorted via recursion.
- ③ merge the sorted parts.

Explain ③ :

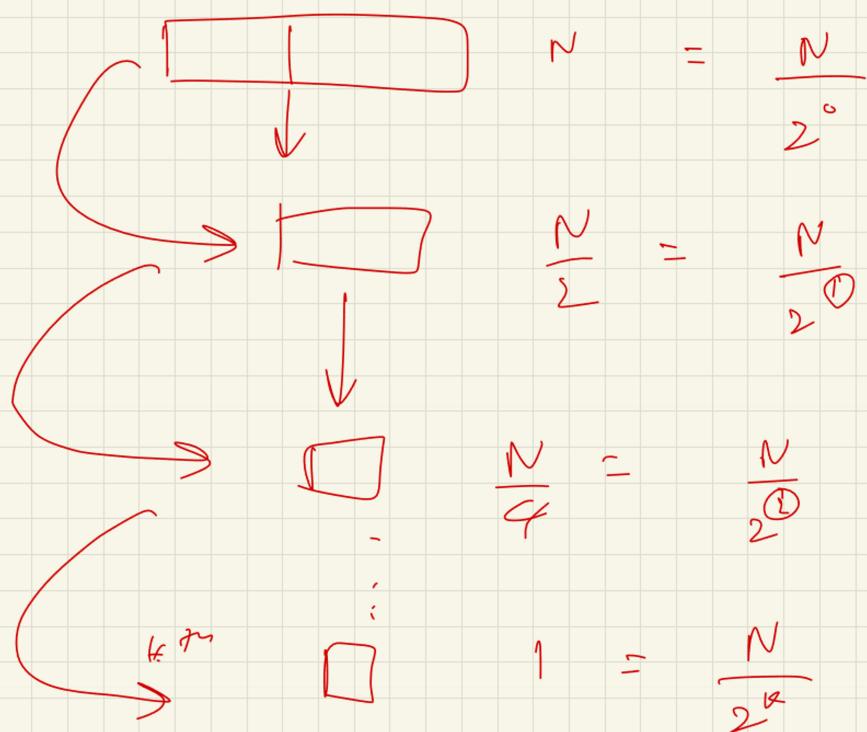


main





\* At every level,  $N$  elements are being merged.



$$1 = \frac{N}{2^k} \Rightarrow 2^k = N$$

$$k \log 2 = \log N$$

$$k = \log_2 N$$

$$\therefore O(N + \log N)$$

Sparse Complex  
Ans  $\propto \mathcal{O}(N)$

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + (n-1)$$

$$= 2T\left(\frac{n}{2}\right) + (n-1)$$

$$2 \times \frac{1}{2} = 1$$

$$\rho = 1$$

$$\int_1^n \frac{1}{u} - \frac{1}{u^2}$$

$$T(n) = n + \alpha \int_1^n \frac{u-1}{u^2}$$

$$= \int \frac{du}{u} - \int \frac{du}{u^2}$$

$$= \log u - \int u^{-2} du$$

$$= \log u + u^{-1}$$

$$= \left[ \log u + \frac{1}{u} \right]_1^n$$

$$= \log n + \frac{1}{n} - 1$$

$$= n + n \left[ \log n + \frac{1}{n} - 1 \right]$$

~~$$= n + n \log n + 1 - n$$~~

Please wait while OneNote loads this Printout...



✗

Please wait while OneNote loads this Printout...



✗

Please wait while OneNote loads this Printout...



X

14 June 2022 13:02

# Recursion Subset, Subsequence, String Questions

14 June 2022 13:03

## Questi

### on : skip a character

- suppose word is given "cat"
- You have to print "ct" skipping 'a' .

```
package com.String2;
```

```
public class Stream {  
    public static void main(String[] args) {  
        skip("", "roshaan");  
    }
```

```
    static void skip(String p, String up){  
        if(up.isEmpty()) {  
            System.out.println(p);  
            return;  
        }  
  
        char ch = up.charAt(0);  
        if(ch == 'a'){  
            skip(p, up.substring(1));  
        }else {  
            skip(p+ch, up.substring(1));  
        }  
    }
```

```
    static String skip2(String up){  
        if(up.isEmpty()) return "";  
        char ch = up.charAt(0);  
        if(ch=='a'){  
            return skip2(up.substring(1));  
        }else{  
            return ch+skip2(up.substring(1));  
        }  
    }
```

```
}
```

➤ Lets see how way 1 works

```
Ch = 'c'  
Ch == 'a' ---> false  
|  
---> skip("c", "at")
```

```
Ch = 'a'  
Ch == 'a' ---> true ---> skip ("c", t)
```

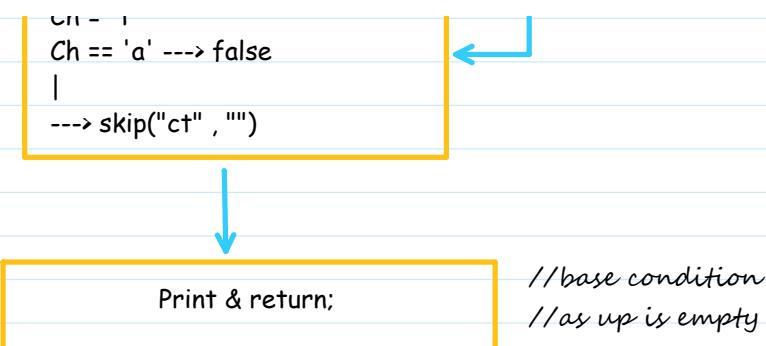
```
Ch = 't'  
Ch == 'a' ---> false  
|
```

//there is 2 way to do  
//lets see how internally it is working

WAY 1

WAY 2

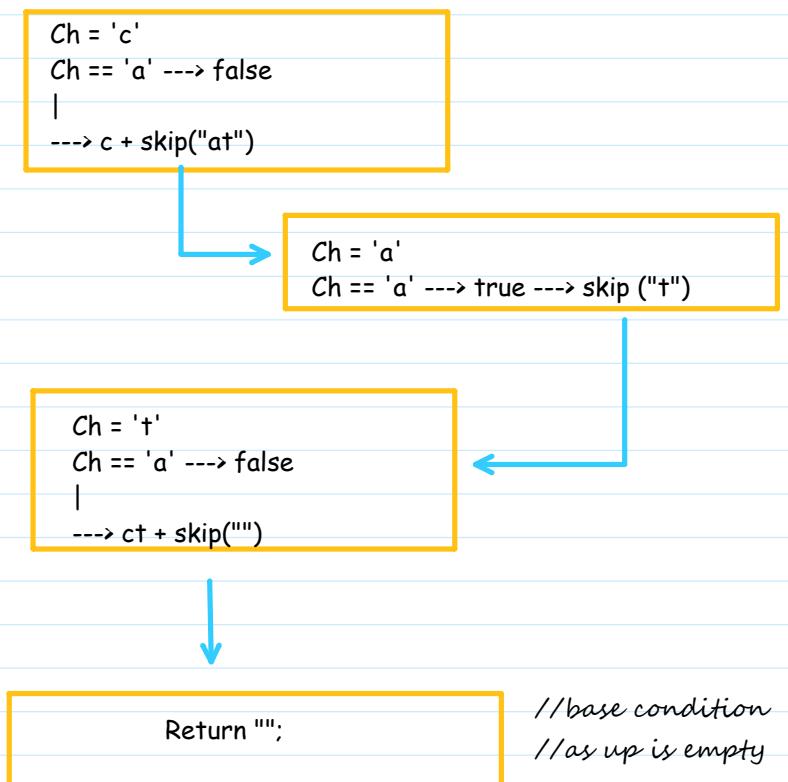
//function of way1  
static void skip(String p, String up){  
 if(up.isEmpty()) {  
 System.out.println(p);  
 return;  
 }  
  
 char ch = up.charAt(0);  
 if(ch == 'a') {



```

char ch = up.charAt(0);
if(ch == 'a'){
    skip(p, up.substring(1));
} else {
    skip(p+ch, up.substring(1));
}
    
```

➤ Lets see how way 2 works



```

//function of way2
static String skip2(String up){
    if(up.isEmpty()) return "";
    char ch = up.charAt(0);
    if(ch=='a'){
        return skip2(up.substring(1));
    } else{
        return ch+skip2(up.substring(1));
    }
}
    
```

Question : skip a String

```

package com.String2;

public class SkipString {
    public static void main(String[] args) {
        System.out.println(skipWord("roshan favorite fruit is applemango"));
    }
    static String skipWord(String up){
        if(up.isEmpty()) return "";
        if(up.startsWith("apple")){
            return skipWord(up.substring(5));
        } else{
            return up.charAt(0) + skipWord(up.substring(1));
        }
    }
}
    
```

o/p : roshan favorite fruit is mango

Question : Skip a string when if it's not a required string.

```
//Skip a string when if it's not a required string.  
//skip app if it's not apple  
static String skipAppNotApple(String up){  
    //base condition  
    if(up.isEmpty()) return "";  
  
    if(up.startsWith("app") && !up.startsWith("apple")){  
        return skipAppNotApple(up.substring(3));  
    }  
    else{  
        return up.charAt(0) + skipAppNotApple(up.substring(1));  
    }  
}
```

- **Subsets**: Non-adjacent collection

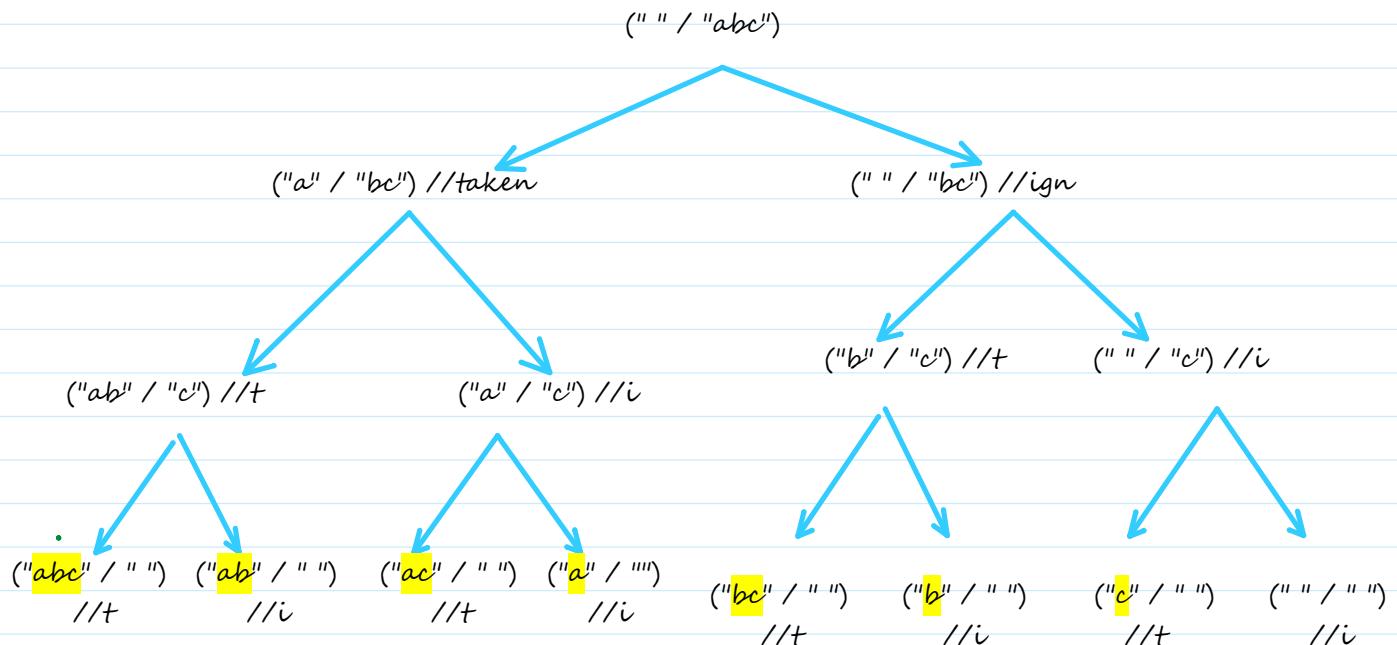
- [3, 5, 9] → it can be → [3], [3,5], [3,9], [3,5,9], [5,9], [5], [9]

- Str = "abc"

Ans = ["a", "b", "c", "ab", "ac", "bc", "abc"]

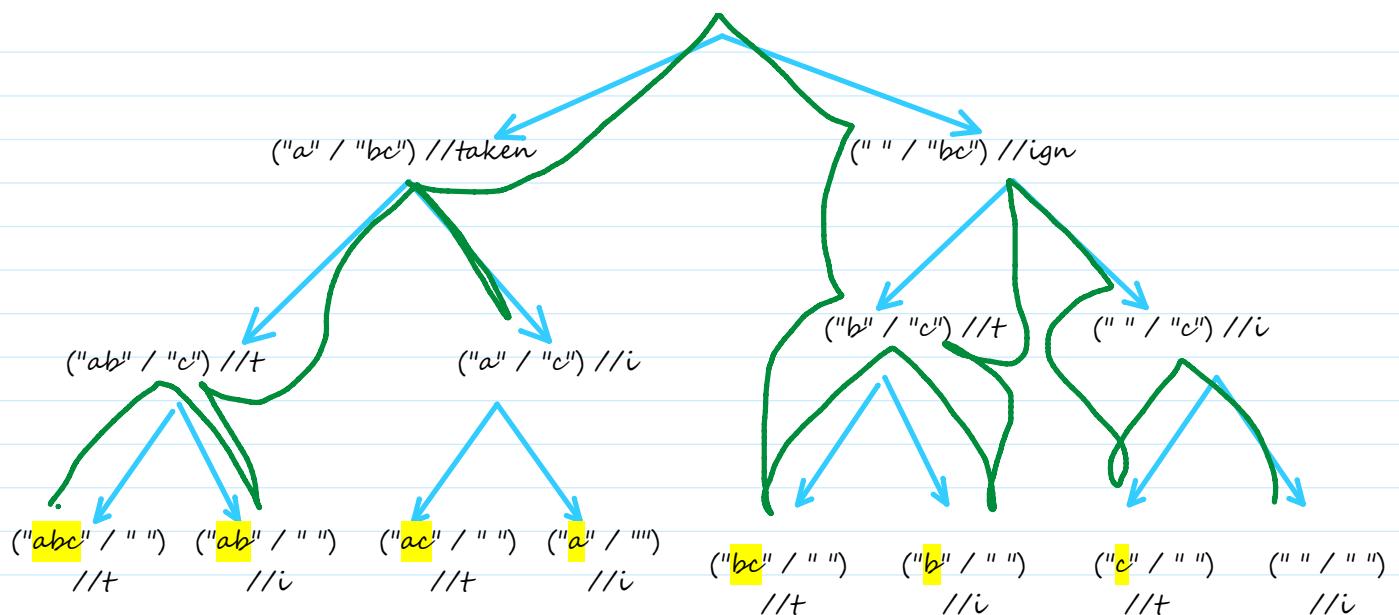
//remember --> it can't be cb bcz order can be changed.

- This pattern of taking some element & removing some is known as this subset pattern.
- You have two option take it or ignore it.



- We call Subset for array
- We call Subsequence for string

( " " / "abc" )



```

package com.RecQue;

import java.util.ArrayList;

public class Temp {
    public static void main(String[] args) {
        String str="abc";
        subseqAscii(str,"");
    }
}

static void subseq(String up, String p) {
    if(up.isEmpty()){
        System.out.println(p);
        return;
    }
    char ch=up.charAt(0);
    subseq(up.substring(1), ch+p); //taken
    subseq(up.substring(1), p); //ignored
}

//using ArrayList
static ArrayList<String> subseq2(String up, String p) {
    if(up.isEmpty()){
        ArrayList<String> list= new ArrayList<>();
        list.add(p);
        return list;
    }
    char ch=up.charAt(0);
    ArrayList<String> left = subseq2(up.substring(1), ch+p); //taken
    ArrayList<String> right = subseq2(up.substring(1), p); //ignored
    left.addAll(right);
    return left;
}

static void subseqAscii(String up, String p) {
    if (up.isEmpty()){
        System.out.println(p);
        return;
    }
    char ch = up.charAt(0);
}

```

- In every order there is constant two branch.

```

//in Ascii, we have to make 3
subseqAscii( up.substring(1), p + ch);
subseqAscii( up.substring(1), p);
subseqAscii( up.substring(1), p + (ch+0));
}

static ArrayList<String> subseqAsciiRet(String up, String p) {
    if (up.isEmpty()) {
        ArrayList<String> list = new ArrayList<>();
        list.add(p);
        return list;
    }
    char ch = up.charAt(0);
    ArrayList<String> first = subseqAsciiRet(up.substring(1), p + ch);
    ArrayList<String> second = subseqAsciiRet(up.substring(1), p);
    ArrayList<String> third = subseqAsciiRet(up.substring(1), p + (ch+0));

    first.addAll(second);
    first.addAll(third);
    return first;
}
}

```

- Iterative way to print

```

package com.String2;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class SubSet
{
    public static void main(String[] args) {
        int arr[]={1,2,3};
        System.out.println(subset(arr));

    }
    static List<List<Integer>> subset(int[] arr){
        List<List<Integer>> outer = new ArrayList<>();

        outer.add((new ArrayList<>()));

        for (int num : arr){
            int n =outer.size();
            for (int i =0; i<n; i++){
                List<Integer> internal = new ArrayList<>(outer.get(i));
                internal.add(num);
                outer.add(internal);
            }
        }
        return outer;
    }
}

```

```

package com.String2;

import java.util.Scanner;

public class SubSet
{
    //To find all the subsets of a string
    static void subString(char str[], int n)
    {
        // To select starting point
        for (int t = 1; t <= n; t++)
        {
            // To select ending point
            for (int i = 0; i <= n - t; i++)
            {
                // Print characters from selected
                // starting to end point.
                int j = i + t - 1;
                for (int k = i; k <= j; k++)
                {
                    System.out.print(str[k]);
                }

                System.out.println();
            }
        }
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        //Take input from the user
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the string is ");
        String str1=sc.nextLine();
        char str[] = str1.toCharArray();
        System.out.println("All the substrings of the above string are: ");
        subString(str, str.length());
    }
}

```

- Subset of a array with duplicate element

```

static List<List<Integer>> subsetDuplicate(int[] arr) {
    //when you find a duplicate element, only add it in the newly created subset of previous step
    //Because of above point duplicate have to be together ---- to bring together we have to sort the array
    Arrays.sort(arr);
    List<List<Integer>> outer = new ArrayList<>();
    outer.add(new ArrayList<>());
    int start = 0;
    int end = 0;
    for (int i = 0; i < arr.length; i++) {
        start = 0;
        // if current and previous element is same, s = e + 1
        if (i > 0 && arr[i] == arr[i-1]) {
            start = end + 1;
        }
        end = outer.size() - 1;
    }
}

```

```
int n = outer.size();
for (int j = start; j < n; j++) {
    List<Integer> internal = new ArrayList<>(outer.get(j));
    internal.add(arr[i]);
    outer.add(internal);
}
return outer;
}
```

# Permutations

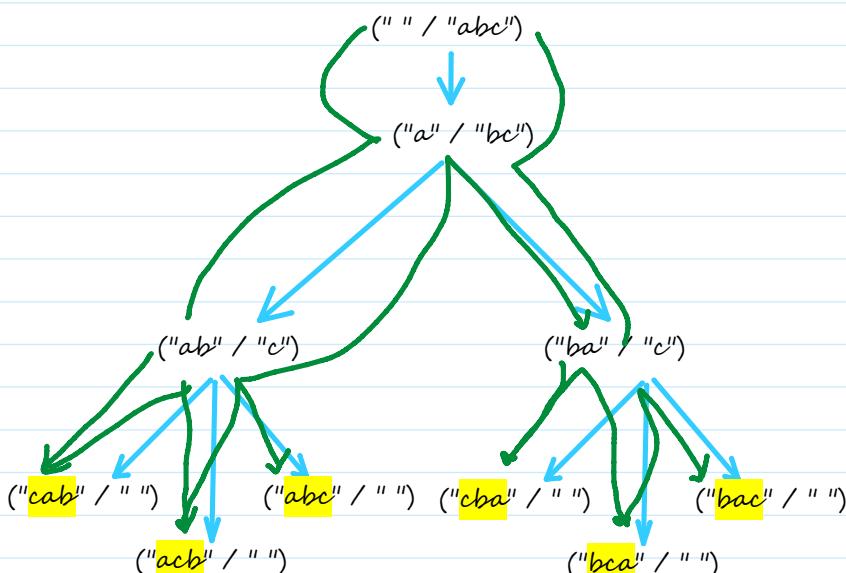
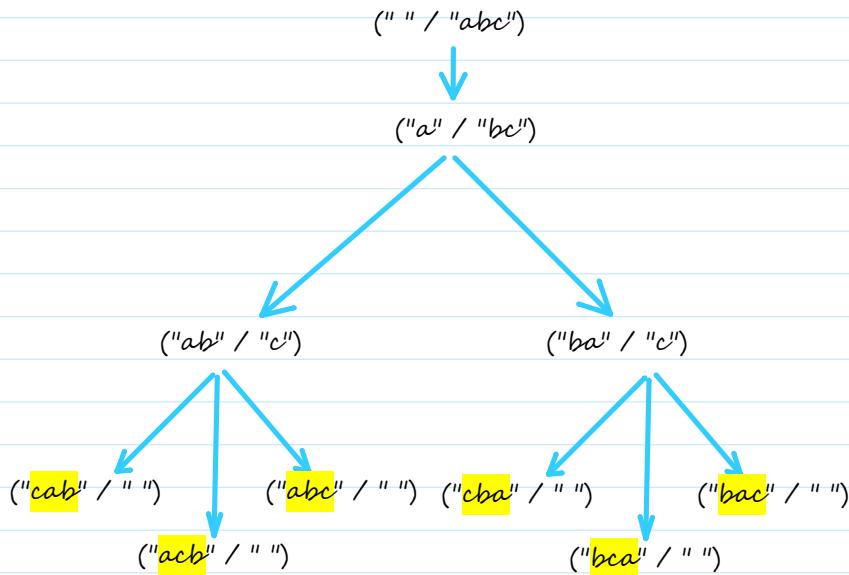
18 June 2022 20:22

- **What is a Permutation?**

A permutation is a mathematical technique that determines the number of possible arrangements in a set when the order of the arrangements matters.

- $abc \rightarrow cba, bac,$  and etc.

$ab (X) \rightarrow$  You cannot skip the character or number



```
package com.RecPerm;
```

```
import java.util.ArrayList;
```

```
public class Perm1 {
```

```

public static void main(String[] args) {
    System.out.println(permuntationCount("", "abc"));
}

static void permuntation(String p, String up) {
    //base condition
    if(up.isEmpty()){
        System.out.println(p);
        return;
    }

    char ch = up.charAt(0);
    for (int i = 0; i <=p.length(); i++) {
        String f = p.substring(0,i);
        String s = p.substring(i,p.length());
        permuntation(f+ch+s, up.substring(1));
    }
}

static ArrayList<String> permuntationAL(String p, String up) {
    //base condition
    if(up.isEmpty()){
        ArrayList<String> list= new ArrayList<>();
        list.add(p);
        return list;
    }
    ArrayList<String> ans = new ArrayList<>();
    char ch =up.charAt(0);
    for (int i = 0; i <=p.length(); i++) {
        String f=p.substring(0,i);
        String s=p.substring(i,p.length());
        ans.addAll(permuntationAL(f+ch+s,up.substring(1)));
    }
    return ans;
}

static int permuntationCount(String p, String up) {
    //base condition
    if(up.isEmpty()){
        return 1;
    }

    int count=0;
    char ch = up.charAt(0);
    for (int i = 0; i <=p.length(); i++) {
        String f = p.substring(0,i);
        String s = p.substring(i,p.length());
        count+=permuntationCount(f+ch+s, up.substring(1));
    }
    return count;
}

```

# Recursion Questions

19 June 2022 16:15

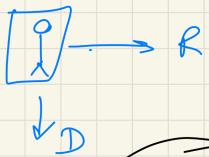
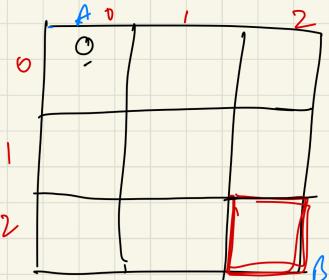
# BACK TRACKING

24 June 2022 02:44

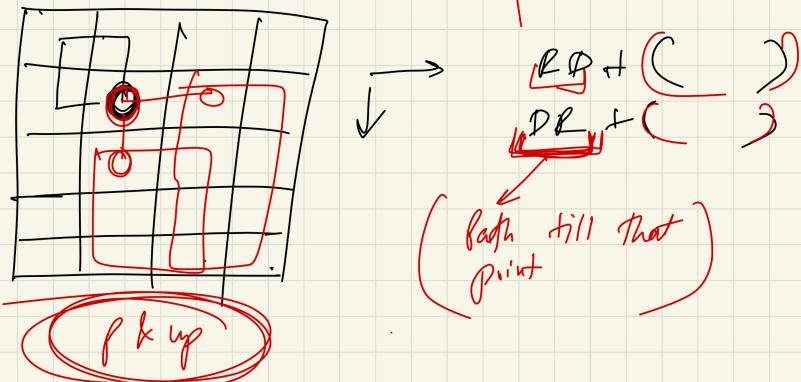
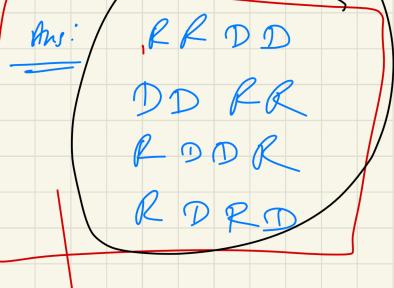


Backtracki...

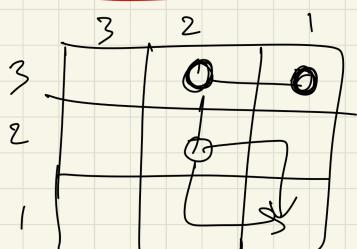
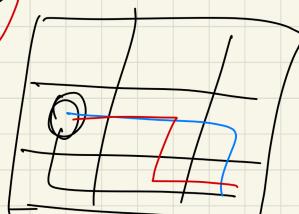
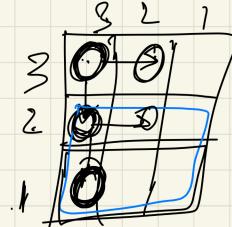
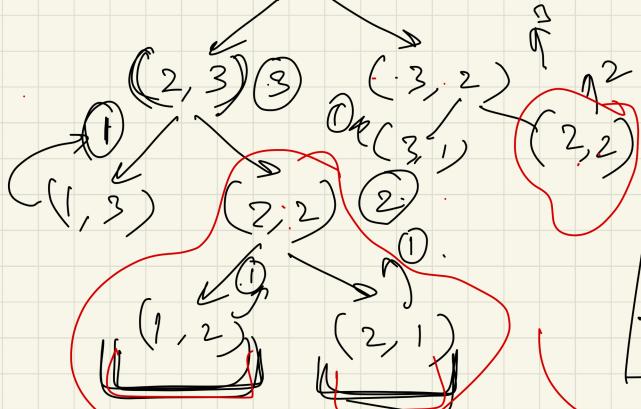




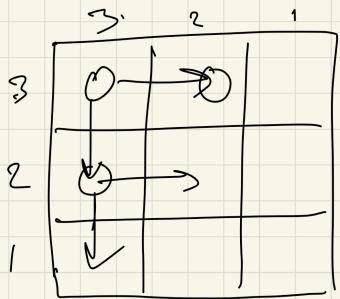
$(P, r, c)$



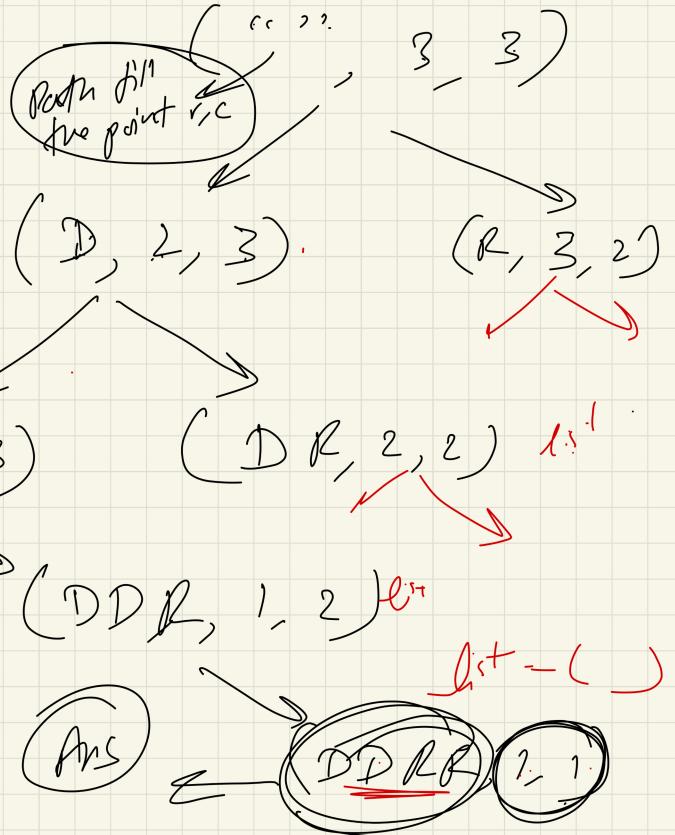
$(r, c)$  (d)



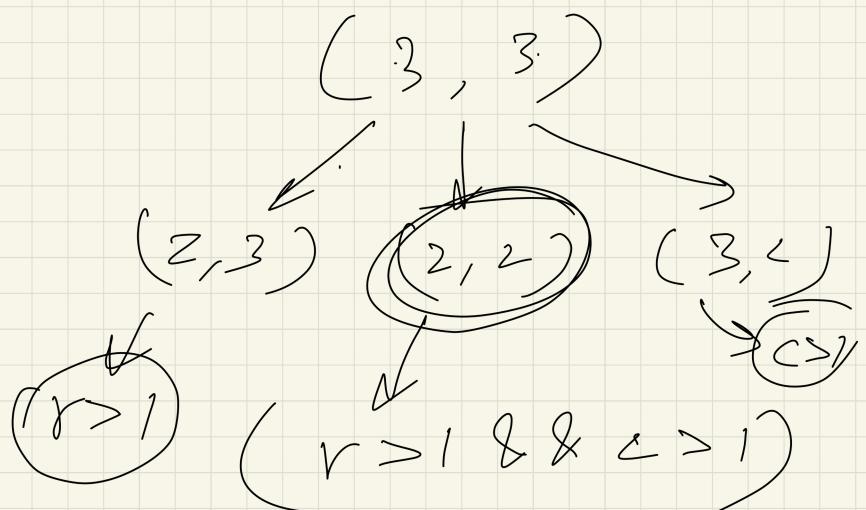
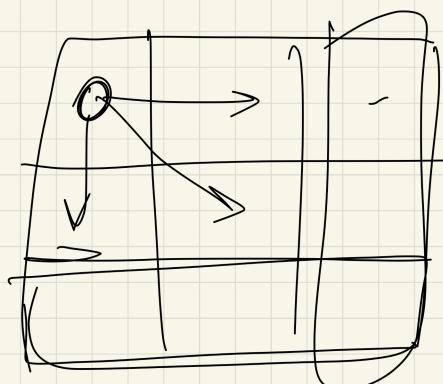
DP future videos



$\rightarrow R$   
 $\downarrow D$

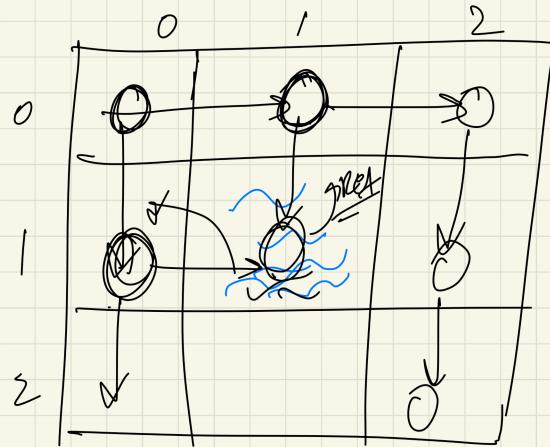


Exactly known  
some as  
subset of  $\{P, UP\}$   
it is not matter  
whether it is forward or backward.

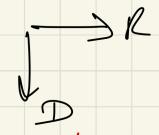


Q:

Maze with obstacles :

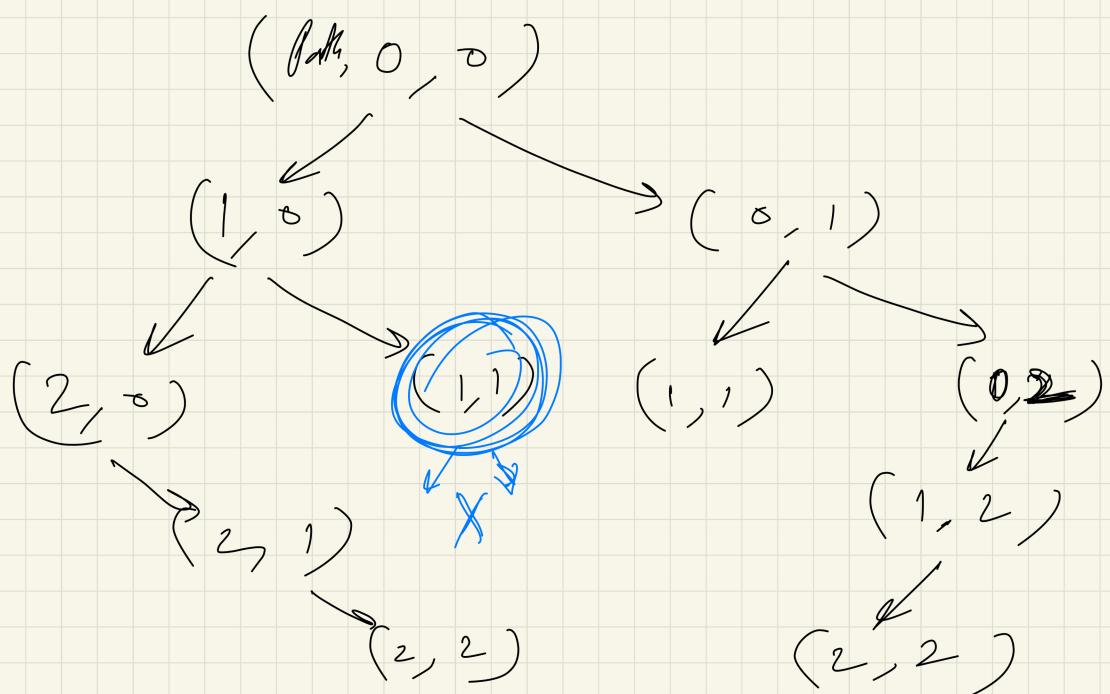


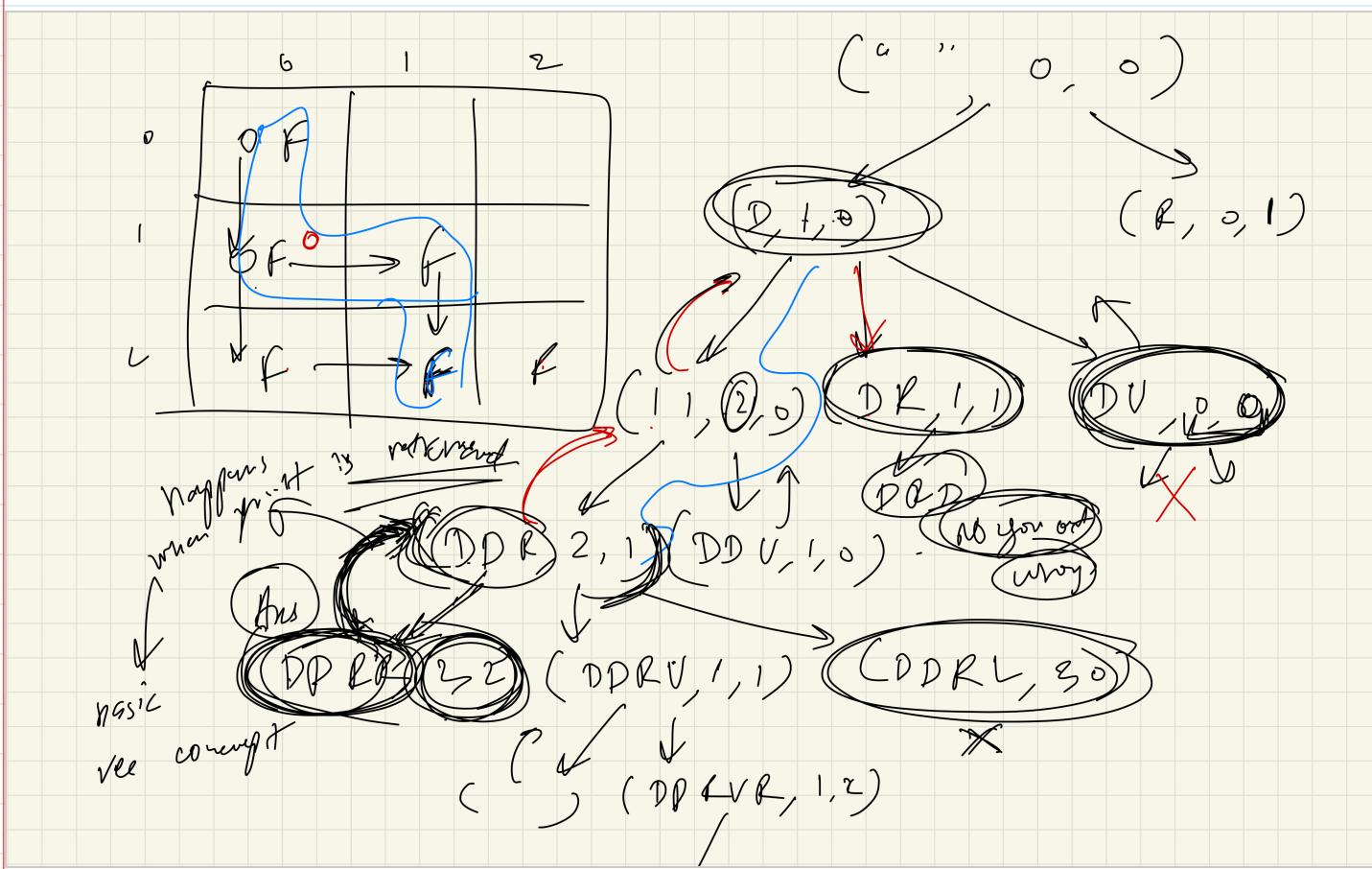
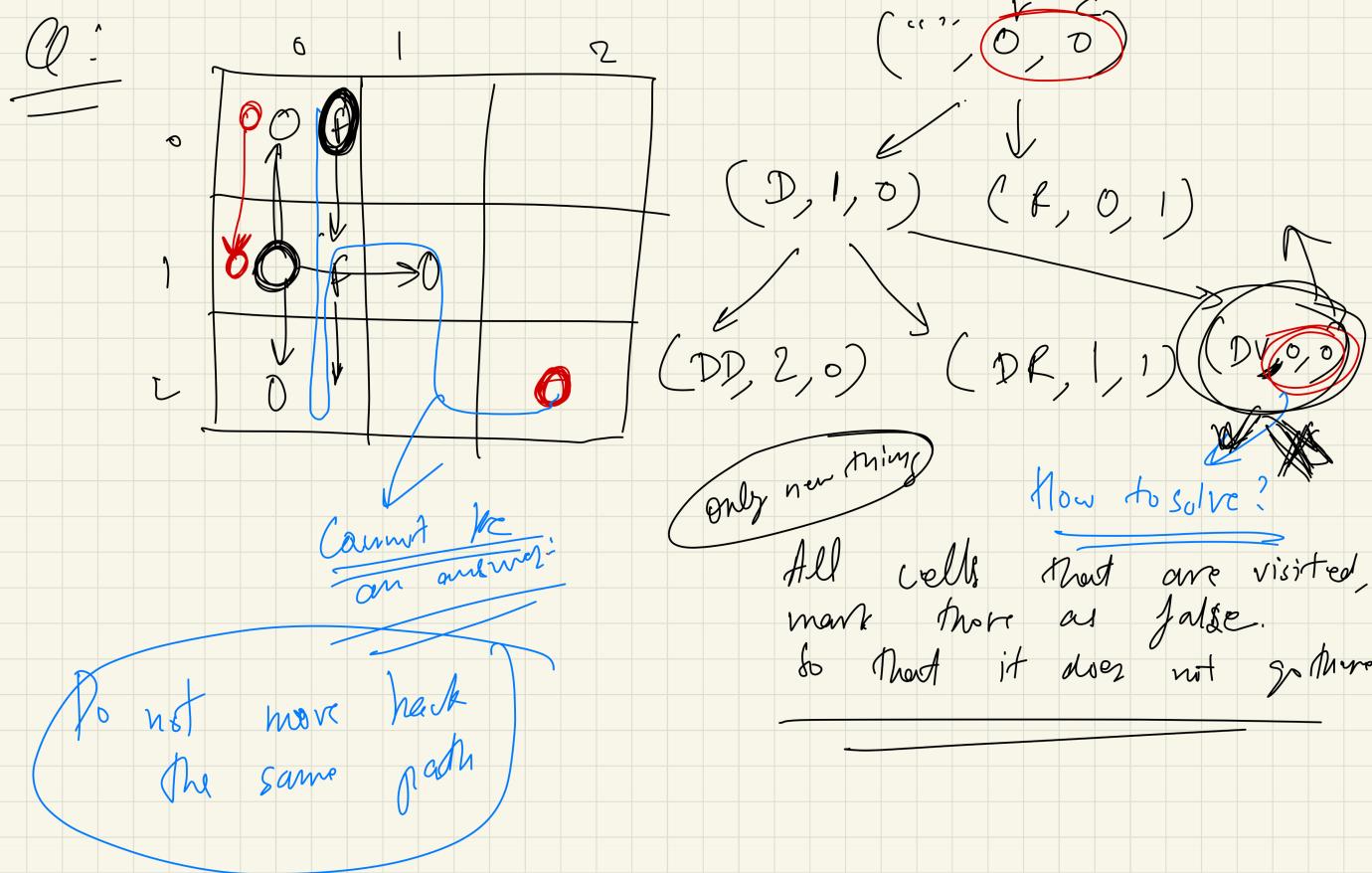
BFS can matrix  
false  $\rightarrow$  River



Maze : When you land on a new cell, check whether that is river or not.

If you land on river, stop recursion for that cell.





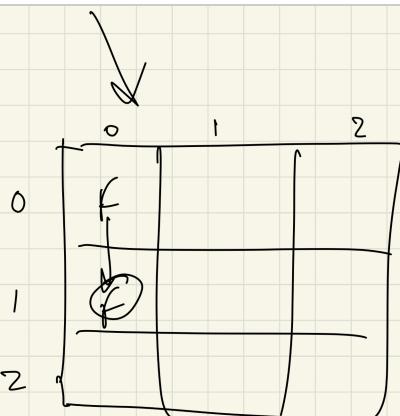
(DPLVRD, ?, ?)

~~Common sense:~~

Moving false == I have that cell  
in my current path.

So when that path is over, ex: you are in  
another recursive call, those cells would not be  
false.

~~While you are moving back, you restore  
the memo as it was.~~



Very first recursion step!

When do we go back?

When the function is returned.

When you come out of the  
recursive function  $\rightarrow$  you are  
now in a new recursive call.

Hence, remember the cell as T.

This is known as  
backtracking.

Q:

1		
2		
3	4	

D D R R

1		
2	S	6
3	4	7

D D R V R D

- \* Take a step variable
- \* Update the path array
- \* Print it if base condition
- \* Backtrack

# BT Problem

26 June 2022 02:09

When to use BT?

When a choice can affect future answer, use BT

- Nqueens

```
package com.BackTracking;

import java.util.ArrayList;

public class NQueen {
    public static void main(String[] args) {
        int n=4;
        boolean board[][]=new boolean[n][n];
        System.out.println(queens(board,0));
    }

    //no of possible ways are there to keep the queen
    static int queens(boolean[][] board, int row) {
        //base condition
        //when the row hit equal to length of board it means count 1 succeed ! ---> in every row we able to keep one
        queen.
        if (row == board.length) {
            display(board);
            System.out.println();
            return 1;
        }

        int count = 0;

        // placing the queen and checking for every row and col
        for (int col = 0; col < board.length; col++) {
            // place the queen if it is safe
            if(isSafe(board, row, col)) {
                board[row][col] = true;
                count += queens(board, row + 1);
                board[row][col] = false;
            }
        }

        return count;
    }

    private static boolean isSafe(boolean[][] board, int row, int col) {
        // check vertical row
        for (int i = 0; i < row; i++) {
            if (board[i][col]) {
                return false;
            }
        }

        // diagonal left
        int maxLeft = Math.min(row, col);
        for (int i = 1; i <= maxLeft; i++) {
            if(board[row-i][col-i]) {
                return false;
            }
        }
```

```

}

// diagonal right
int maxRight = Math.min(row, board.length - col - 1);
for (int i = 1; i <= maxRight; i++) {
    if(board[row-i][col+i]) {
        return false;
    }
}

return true;
}

private static void display(boolean[][] board) {
    for(boolean[] row : board) {
        for(boolean element : row) {
            if (element) {
                System.out.print("Q ");
            } else {
                System.out.print("X ");
            }
        }
        System.out.println();
    }
}
}

```

- Nknights

```

package com.BackTracking;

public class NKnights {
    public static void main(String[] args) {
        int n = 4;
        boolean[][] board = new boolean[n][n];
        knight(board, 0, 0, 4);
    }

    static void knight(boolean[][] board, int row, int col, int knights) {
        if (knights == 0) {
            display(board);
            System.out.println();
            return;
        }

        if (row == board.length - 1 && col == board.length) {
            return;
        }

        if (col == board.length) {
            knight(board, row + 1, 0, knights);
            return;
        }

        if (isSafe(board, row, col)) {
            board[row][col] = true;

```

```

        knight(board, row, col + 1, knights - 1);
        board[row][col] = false;
    }

    knight(board, row, col + 1, knights);
}

private static boolean isSafe(boolean[][] board, int row, int col) {
    if (isValid(board, row - 2, col - 1)) {
        if (board[row - 2][col - 1]) {
            return false;
        }
    }

    if (isValid(board, row - 1, col - 2)) {
        if (board[row - 1][col - 2]) {
            return false;
        }
    }

    if (isValid(board, row - 2, col + 1)) {
        if (board[row - 2][col + 1]) {
            return false;
        }
    }

    if (isValid(board, row - 1, col + 2)) {
        if (board[row - 1][col + 2]) {
            return false;
        }
    }

    return true;
}

// do not repeat yourself, hence created this function
static boolean isValid(boolean[][] board, int row, int col) {
    if (row >= 0 && row < board.length && col >= 0 && col < board.length) {
        return true;
    }
    return false;
}

private static void display(boolean[][] board) {
    for (boolean[] row : board) {
        for (boolean element : row) {
            if (element) {
                System.out.print("K ");
            } else {
                System.out.print("X ");
            }
        }
        System.out.println();
    }
}

```

- Sudoku Solver

```
public static void main(String[] args) {
    int[][] board = new int[][]{
        {3, 0, 6, 5, 0, 8, 4, 0, 0},
        {5, 2, 0, 0, 0, 0, 0, 0, 0},
        {0, 8, 7, 0, 0, 0, 0, 3, 1},
        {0, 0, 3, 0, 1, 0, 0, 8, 0},
        {9, 0, 0, 8, 6, 3, 0, 0, 5},
        {0, 5, 0, 0, 9, 0, 6, 0, 0},
        {1, 3, 0, 0, 0, 0, 2, 5, 0},
        {0, 0, 0, 0, 0, 0, 0, 7, 4},
        {0, 0, 5, 2, 0, 6, 3, 0, 0}
    };

    if (solve(board)) {
        display(board);
    } else {
        System.out.println("Cannot solve");
    }
}

static boolean solve(int[][] board) {
    int n = board.length;
    int row = -1;
    int col = -1;

    boolean emptyLeft = true;

    // this is how we are replacing the r,c from arguments
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i][j] == 0) {
                row = i;
                col = j;
                emptyLeft = false;
                break;
            }
        }
        // if you found some empty element in row, then break
        if (emptyLeft == false) {
            break;
        }
    }

    if (emptyLeft == true) {
        return true;
        // soduko is solved
    }

    // backtrack
    for (int number = 1; number <= 9; number++) {
        if (isSafe(board, row, col, number)) {
            board[row][col] = number;
```

```

        if (solve(board)) {
            // found the answer
            return true;
        } else {
            // backtrack
            board[row][col] = 0;
        }
    }
    return false;
}

private static void display(int[][] board) {
    for(int[] row : board) {
        for(int num : row) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}

static boolean isSafe(int[][] board, int row, int col, int num) {
    // check the row
    for (int i = 0; i < board.length; i++) {
        // check if the number is in the row
        if (board[row][i] == num) {
            return false;
        }
    }

    // check the col
    for (int[] nums : board) {
        // check if the number is in the col
        if (nums[col] == num) {
            return false;
        }
    }

    int sqrt = (int)(Math.sqrt(board.length));
    int rowStart = row - row % sqrt;
    int colStart = col - col % sqrt;

    for (int r = rowStart; r < rowStart + sqrt; r++) {
        for (int c = colStart; c < colStart + sqrt; c++) {
            if (board[r][c] == num) {
                return false;
            }
        }
    }
    return true;
}

```

# BT pdf

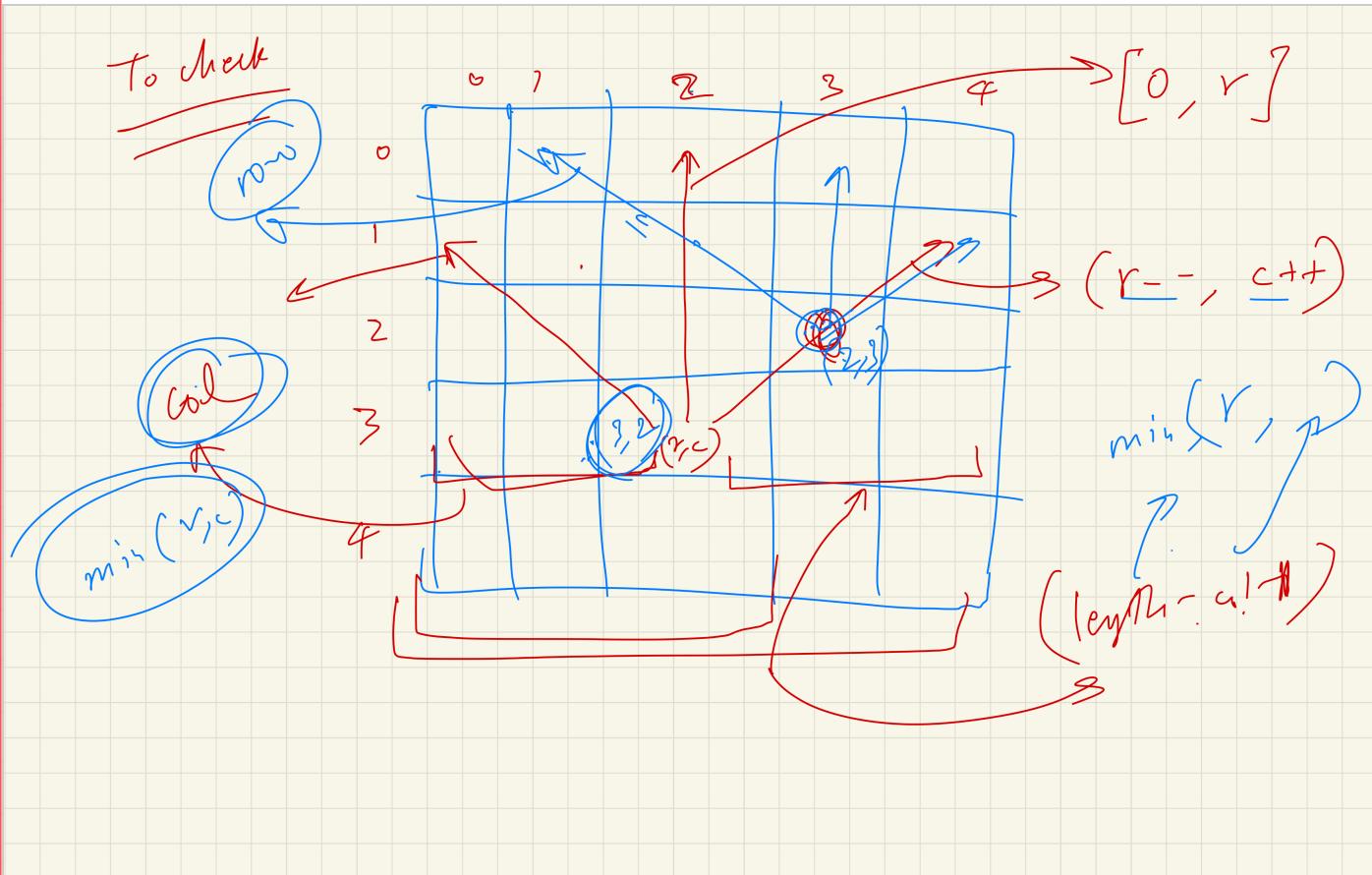
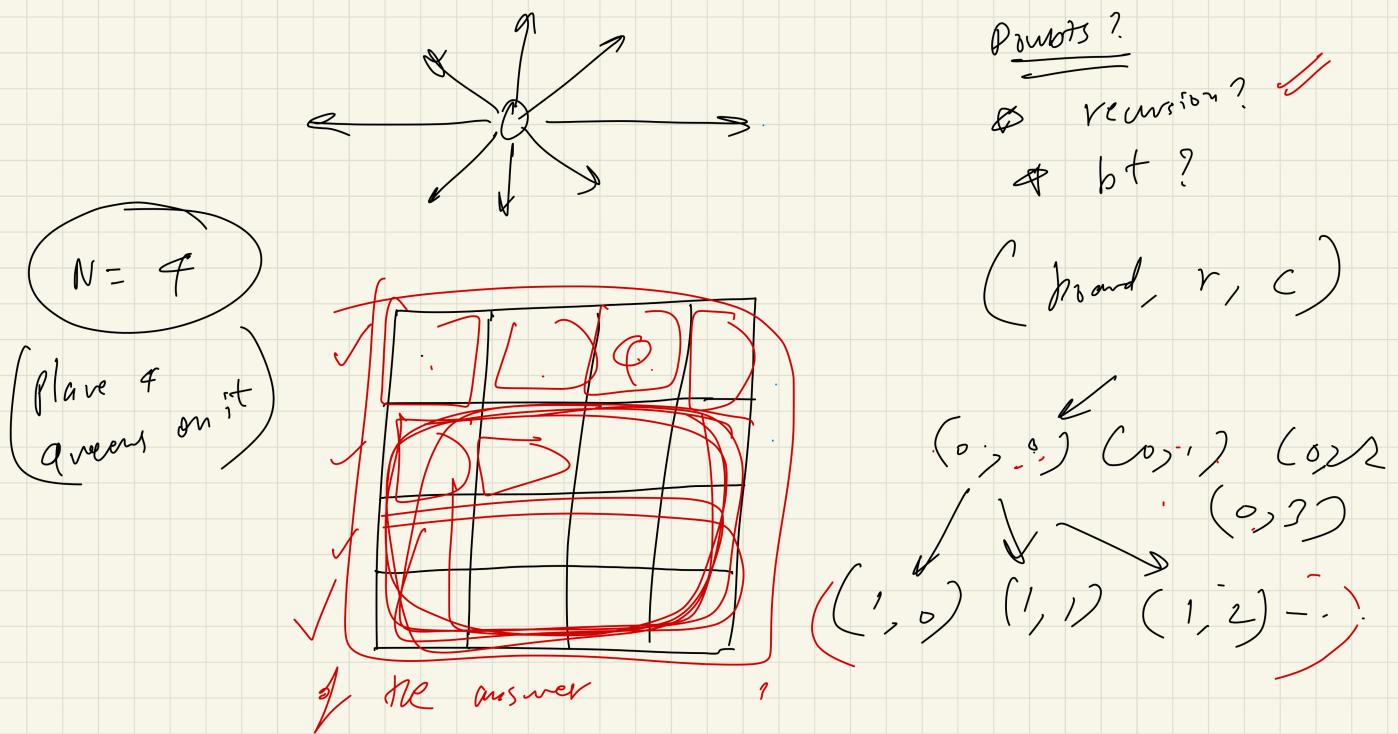
28 June 2022 19:59



Backtracki...  
- Questions



# Q! N-Queens Problem:



Recurrsive Relation:



$(n-1)$

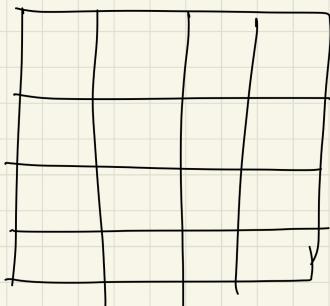
$$T(n) = n * T(n-1) + O(n^2)$$

Aktiver Zeichenfaden:

Time complexity  
lecture.

$$O(n^3 + n!) = \underline{\underline{O(M)}}$$

Ans



$$4 * 3 * 2 * 1$$

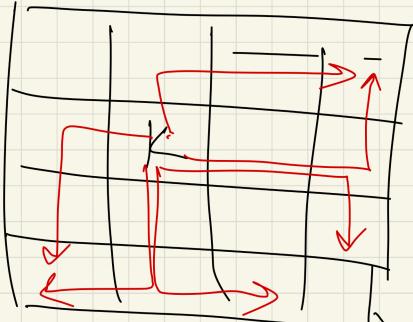
$$= 4!$$

\* You can eliminate  
for loops with  
conditions, but then  
you need another  
variable in ans.

(bound, row, col, targets)  
(g, o, f)

Note:

When a driver function  
answers, use bt.



	0	1	2	3
0	K	K	K	.
1		K	"	"
2	.			
3				

(0, 0, 4)

(0, 1, 3)

(0, 2, 2)

(1, 1, 1)

$r=2, c=1$

$r=2, c=1$

$r=1, c=2$

$r=1, c=2$

0	1	2	3	4	5	6	7	8
0	5	3	1	2	7	6		0
1	6	-	-	1	9	5		
2	9	8					6	
3	8			6				3
4	4			8	3			1
5	7			2				6
6	6				2	8		
7	4	1	9				5	
8		8			7	9		

remainder

(bound)

(6, 2)

(1, 7)

(0, 3)

(3, 6)

$$3 - 3 \times 3 = 0$$

$$6 - 6 \times 3 = 0$$

(6, 8)

$$6 - 6 \times 3 = 0$$

(6,3) ✓

$$7 - \cancel{2 \times 3} = 6$$

$$4 - \cancel{4 \times 3} = 3$$

(6,6)

Complexity : Total q numbers.

for every  $\Theta(n^2)$

f

Time :

$$\Theta(q^{n^2})$$

$n^2$

Space  $\Theta(N^2)$

# Introduction & Concepts - Classes, Objects, Constructors, Keywords

29 June 2022 02:15

## class

A class is a template for an object, and an object is an instance of a class.

A class creates a new data type that can be used to create objects.

When you declare an object of a class, you are creating an instance of that class.

Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

## object

Objects are characterized by three essential properties: state, identity, and behaviour.

The state of an object is a value from its data type. The identity of an object distinguishes one object from another.

It is useful to think of an object's identity as the place where its value is stored in memory.

The behaviour of an object is the effect of data-type operations.

## Dot operator

The dot operator links the name of the object with the name of an instance variable.

Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator.

In simple term you can say by referencing class variable you can access class variable and function.

## New keyword

The 'new' keyword dynamically allocates(that is, allocates at run time)memory for an object & returns a reference to it.

This reference is, more or less, the address in memory of the object allocated by new.

This reference is then stored in the variable.

Thus, in Java, all class objects must be dynamically allocated.

```
Box mybox; // declare reference to object  
mybox = new Box(); // allocate a Box object
```

The first line declares mybox as a reference to an object of type Box. At this point, mybox does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds, in essence, the memory address of the actual Box object.

The key to Java's safety is that you cannot manipulate references as you can actual pointers.

Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

## A Closer Look at new:

```
classname class-var = new classname ( );
```

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

You might be wondering why you do not need to use new for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects.

Rather, they are implemented as "normal" variables.  
This is done in the interest of efficiency.

It is important to understand that new allocates memory for an object during run time.

```
Box b1 = new Box();
Box b2 = b1;
```

b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

```
int square(int i){
    return i * i;
}
```

A parameter is a variable defined by a method that receives a value when the method is called.  
For example,

in square( int i), i is a parameter. An argument is a value that is passed to a method when it is invoked.

For example, square(100) passes 100 as an argument. Inside square( ), the parameter i receives that value.

NOTE:

```
Bus bus = new Bus();
lhs(reference i.e. bus) is looked be compiler & rhs (object i.e. new Bus()) is looked by jvm
```

```
//intro
package com.kunal.introduction;
```

```
import java.sql.Struct;
import java.util.ArrayList;
import java.util.Arrays;

Public class Main{
    public static void main(String[] args) {
        // store 5 roll nos
        int[] numbers= new int[5];

        // store 5 names
        String[] names= new String[5];

        // data of 5 students: {roll no, name, marks}
        int[] rno= new int[5];
        String[] name= new String[5];
        float[] marks= new float[5];
```

```
Student[] students= new Student[5];
```

```
// just declaring
//     Student kunal;
//     kunal = new Student();
```

But here seems to clumsy and  
not in organized way and  
furthermore you can write in  
one line

Student -> class name

kunal -> referencing to class

new -> dynamically allocates (that is, allocates

```

// Kunal = new Student();
Student Kunal= new Student(15, "Kunal Kushwaha", 85.4f);
Student rahul= new Student(18, "Rahul Rana", 90.3f);

// Kunal.rno = 13;
// Kunal.name = "Kunal Kushwaha";
// Kunal.marks = 88.5f;

// Kunal.changeName("Shoe lover");
// Kunal.greeting();

// System.out.println(Kunal.rno);
// System.out.println(Kunal.name);
// System.out.println(Kunal.marks);

Student random= new Student(Kunal);
System.out.println(random.name);

Student random2= new Student();
System.out.println(random2.name);

Student one= new Student();
Student two= one;

one.name= "Something something";

System.out.println(two.name);

}

```

```

// create a class
// for every single student
class Student{
    int rno;
    String name;
    float marks= 90;

    // we need a way to add the values of the above
    // properties object by object
}

// we need one word to access every object

```

```

void greeting(){
    System.out.println("Hello! My name is "+ this.name);
}

void changeName(String name){
    this.name= name;
}

Student(Student other){
    this.name= other.name;
    this.rno= other.rno;
    this.marks= other.marks;
}

```

Kunal -> referencing to class  
new -> dynamically allocates (that is, allocates at run time) memory for an object  
-> it creates 5 objects

(dot) operator is used to access class, structure, or union members.

The `this` keyword refers to the current object in a method or constructor. The most common use of the `this` keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor).

```
this.name= other.name;  
this.rno= other.rno;  
this.marks= other.marks;  
}
```

a class variable  
shadowed by a method  
or constructor  
parameter).

```
Student(){  
    // this is how you call a constructor from another constructor  
    // internally: new Student(13, "default person", 100.0f);  
    this(13, "default person", 100.0f);  
}
```

```
// Student arpit = new Student(17, "Arpit", 89.7f);  
// here, this will be replaced with arpit
```

```
Student(int rno, String name, float marks){  
    this.rno= rno;  
    this.name= name;  
    this.marks= marks;  
}
```

```
}
```

```
//wrapper class
```

```
package com.kunal.introduction;
```

```
public class WrapperExample {  
    public static void main(String[] args) {  
        // int a = 10; //prim data type  
        // int b = 20;  
        //  
        Integer num = 45; //non Prim Data type ---> you can access java inbuild function by using wrapper class
```

```
Integer a = 10;
```

```
Integer b = 20;
```

```
swap(a, b);
```

```
System.out.println(a + " " + b);
```

```
// final int bonus = 2;  
// bonus = 3;
```

```
final A kunal = new A("Kunal Kushwaha");  
kunal.name = "other name";
```

```
// when a non primitive is final, you cannot reassign it.  
// kunal = new A("new object");
```

```
A obj = new A("Rnadvsjhv");
```

```
System.out.println(obj);
```

```
// for (int i = 0; i < 1000000000; i++) {  
//     obj = new A("Random name");  
// }
```

```
}
```

```
static void swap(Integer a, Integer b) {
    Integer temp = a;
    a = b;
    b = temp;
}

class A {
    final int num = 10;
    String name;

    public A(String name) {
        // System.out.println("object created");
        this.name = name;
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Object is destroyed");
    }
}
```

- Packages

Packages are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

The package is both a naming and a visibility control mechanism.

The following statement creates a package called My Package: package My Package;

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called My Package. Remember that case is significant, and the directory name must match the package name exactly.

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully.

You cannot rename a package without renaming the directory in which the classes are stored.

How does the Java run-time system know where to look for packages that you create?

The answer has three parts:

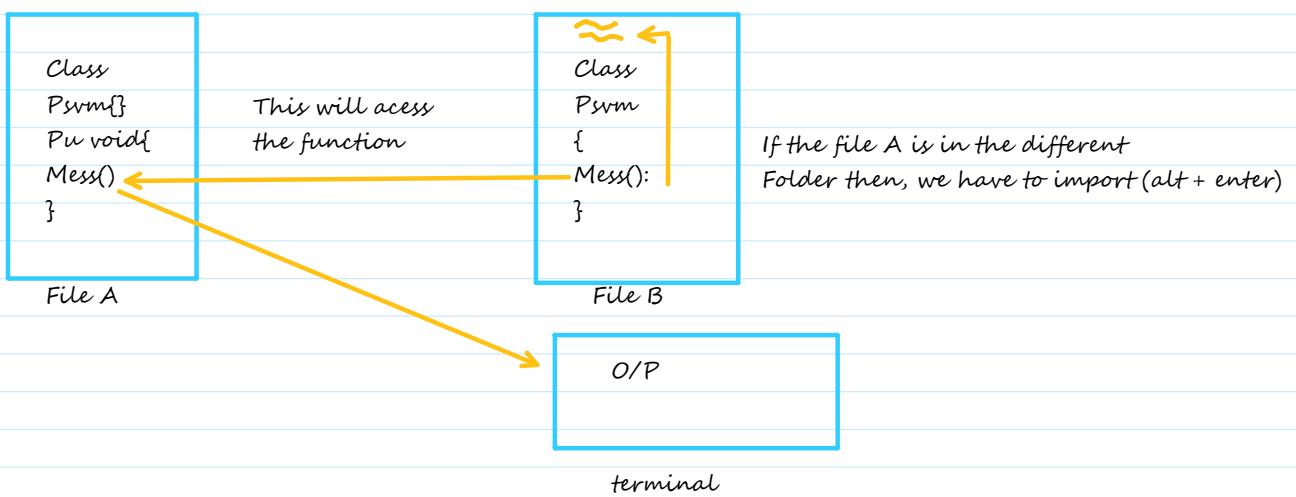
- First, by default, the Java run-time system uses the current working directory as its starting point.

Thus, if your package is in a subdirectory of the current directory, it will be found.

- Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.

- Third, you can use the -classpath option with java and javac to specify the path to your classes.

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.



NOTE : when the file are in same folder then you don't have to import it.

- Static

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static.

The most common example of a static member is main(). main() is declared as static because it must be called before any objects exist.

Static method in Java is a method which belongs to the class and not to the object.

A static method can access only static data. It cannot access non-static data (instance variables). A non-static member belongs to an instance. It's meaningless without somehow resolving which instance of a class you are talking about. In a static context, you don't have an instance, that's why you can't access a non-static member without explicitly mentioning an object reference.

In fact, you can access a non-static member in a static context by specifying the object reference explicitly :

```
public class Human {  
  
    String message = "Hello World";  
  
    public static void display(Human human){  
        System.out.println(human.message);  
    }  
  
    public static void main(String[] args) {  
        Human kunal = new Human();  
        kunal.message = "Kunal's message";  
        Human.display(kunal);  
    }  
  
}
```

A static method can call only other static methods and cannot call a non-static method from it.

A static method can be accessed directly by the class name and doesn't need any object.

A static method cannot refer to "this" or "super" keywords in anyway.

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

```
// Demonstrate static variables, methods, and blocks.  
class UseStatic {  
    static int a = 3;  
    static int b;  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
    static {  
        System.out.println("Static block initialized.");  
        b = a * 4;  
    }  
    public static void main(String args[]) {  
        meth(42);  
    }  
}
```

```
}
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to  $a^4$  or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Here is the output of the program:

Static block initialized.

x = 42

a = 3

b = 12

Note: main method is static, since it must be accessible for an application to run, before any instantiation takes place.

NOTE: Only nested classes can be static.

NOTE: Static inner classes can have static variables

You can't override the inherited static methods, as in java overriding takes place by resolving the type of object at run-time and not compile time, and then calling the respective method.

Static methods are class level methods, so it is always resolved during compile time.

Static INTERFACE METHODS are not inherited by either an implementing class or a sub-interface.

NOTE:

```
public class Static {
```

```
// class Test // ERROR
```

```
static class Test{
```

```
    String name;
```

```
    public Test(String name) {
```

```
        this.name = name;
```

```
}
```

```
}
```

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

```
        Test a = new Test("Kunal");
```

```
        Test b = new Test("Rahul");
```

```
        System.out.println(a.name); // Kunal
```

```
        System.out.println(b.name); // Rahul
```

```
}
```

```
}
```

Because :

The static keyword may modify the declaration of a member type C within the body of a non-inner class or interface T.

Its effect is to declare that C is not an inner class. Just as a static method of T has no current instance of T in its body, C also has no current instance of T, nor does it have any lexically enclosing instances.

Here, test does not have any instance of its outer class Static. Neither does main.

But main & Test can have instances of each other.

```
//sample program to understand
package com.OOP2;
```

```

public class Human {
    void grett(){
        System.out.println("hello");
    }
    static class Student{
        static int rn;
        String name;
        int std;

        public Student(int rn, String name, int std) {
            this.rn = rn;
            this.name = name;
            this.std = std;
        }
    }
}

public static void main(String[] args) {
    Human b = new Human();
    // a.Student(24,"sai",4);
    b.grett();
    Human.Student.rn=44;

    Student.rn=24; //now it's fine bcz rn is static! ---> it will throw error when it will non-static.
/*
we are making the object, although it is static class bcz the given variable & function
inside the class which is non-static to access them we are making object of a whole class
--- so now by using this object we can access the function & variable.
*/
    Student a = new Student(24,"roshan",4);
}

```

- Singleton
- Singleton is a design pattern that ensures that a class can only have one object.

<pre> package com.OOP2;  public class Singleton {     //private --&gt; if there is private     //then it will only allow to run in that class.      private Singleton() {     }      //create an object of Singleton     private static Singleton instance;      //as it is allowed to make object in this class.     public static Singleton getInstance() {         if (instance == null) {             instance = new Singleton();         }         return instance;     } } </pre>	<pre> package com.OOP2;  public class Main {     public static void main(String[] args) {         // Singleton obj = new Singleton();         //throwing error          // we are able to make the object ---&gt; just         //for one time.         Singleton obj1= Singleton.getInstance();         // 2.         Singleton obj2= Singleton.getInstance();         // 3.         Singleton obj3= Singleton.getInstance();          // and other objects are pointing to first         //object only     } } </pre>
---	--

Singleton class

Main class

# OOP3

19 July 2022 14:06

To inherit a class, you simply incorporate the definition of one class into another by using the `extends` keyword.

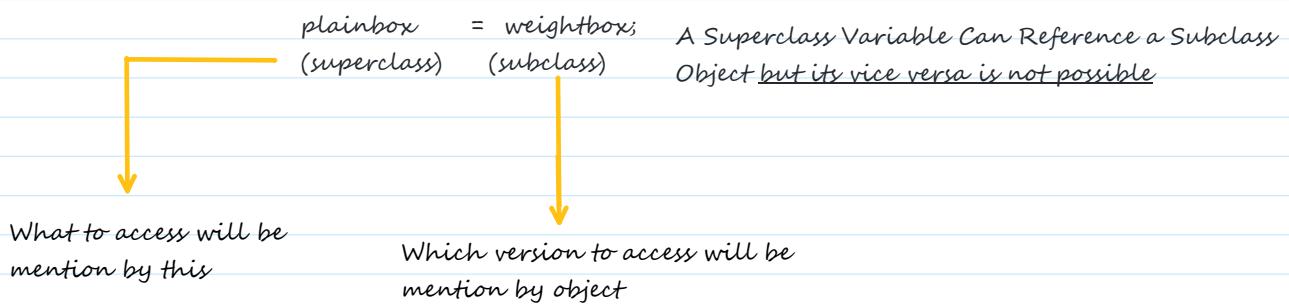
```
class subclass-name extends superclass-name {  
    // body of class  
}
```

It will use the properties of own class plus the properties of his parent class

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple super classes into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

A Superclass Variable Can Reference a Subclass Object: It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.



```
package com.OOP3;  
  
class A{  
    public void show(){  
        System.out.println("A");  
    }  
}  
class B extends A{  
    public void show(){  
        System.out.println("B");  
    }  
}  
public class Temp {  
    public static void main(String[] args) {  
        A obj = new B();  
        obj.show(); // It will print B  
        // B obj1 = new A(); -----> It will throw error  
    }  
}
```

SUPERCLASS ref = new SUBCLASS(); // HERE ref can only access methods which are available in SUPERCLASS

➤ Super

The super keyword refers to superclass (parent) objects.

```
public class Temp {  
    double height;  
    double width;  
    double length;  
  
    public Temp(double height, double width, double length)  
    {  
        this.height = height;  
        this.width = width;  
        this.length = length;  
    }  
}
```



```
public class Temp2 extends Temp {  
    double weight;  
  
    Temp2(double height, double width, double length,  
          double weight) {  
        super(height, width, length);  
        this.weight = weight;  
    }  
}
```



Simply referring to above box (super class).

Super has no idea about sub class.

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword `super`.

`super` has two general forms. The first calls the superclass' constructor.

The second is used to access a member of the superclass that has been hidden by a member of a subclass.

```
BoxWeight(double w, double h, double d, double m) {  
    super(w, h, d); // call superclass constructor  
    weight = m;  
}
```

Here, `BoxWeight()` calls `super()` with the arguments `w`, `h`, and `d`. This causes the `Box` constructor to be called, which initializes `width`, `height`, and `depth` using these values. `BoxWeight` no longer initializes these values itself. It only needs to initialize the value unique to it: `weight`. This leaves `Box` free to make these values private if desired.

Thus, `super()` always refers to the superclass immediately above the calling class. This is true even in a multilevel hierarchy.

```
class Box {  
    private double width;  
    private double height;  
    private double depth;
```

```

// construct clone of an object

Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}

class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object

    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }
}

```

Notice that `super()` is passed an object of type `BoxWeight`—not of type `Box`. This still invokes the constructor `Box(Box ob)`.

NOTE: A superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a `BoxWeight` object to the `Box` constructor. Of course, `Box` only has knowledge of its own members.

### A Second Use for `super`

The second form of `super` acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.

`super.member`

Here, `member` can be either a method or an instance variable. This second form of `super` is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. `super( )` always refers to the constructor in the closest superclass. The `super( )` in `BoxPrice` calls the constructor in `BoxWeight`. The `super( )` in `BoxWeight` calls the constructor in `Box`. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a subclass needs parameters of its own.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

NOTE: If `super( )` is not used in subclass' constructor, then the default or parameterless constructor of each superclass will be executed.

## ➤ Inheritance

### 1. Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, `Dog` class inherits the `Animal` class, so there is the single inheritance.

```

class Animal{
    void eat(){
        System.out.println("eating...");}
}

```

```

class Dog extends Animal{
void bark(){
{System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}

```

Output:  
barking...  
eating...

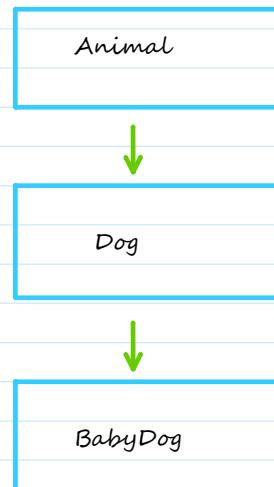
## 2. Multilevel Inheritance

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```

class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){
System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){
System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}

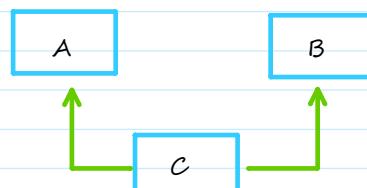
```



Output:  
weeping...  
barking...  
eating...

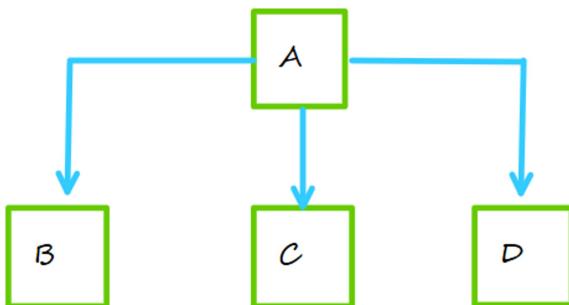
## 3. Multilevel Inheritance

It is not supported by java bcz The reason behind this is to prevent ambiguity. Consider a case where class B extends class A and Class C and both class A and C have the same method display(). Now java compiler cannot decide, which display method it should inherit. To prevent such situation, multiple inheritances is not allowed in java.



#### 4. Hierarchical Inheritance

One class is inherited by many classes.



```
class A {  
    public void print_A() { System.out.println("Class A"); }  
}
```

```
class B extends A {  
    public void print_B() { System.out.println("Class B"); }  
}
```

```
class C extends A {  
    public void print_C() { System.out.println("Class C"); }  
}
```

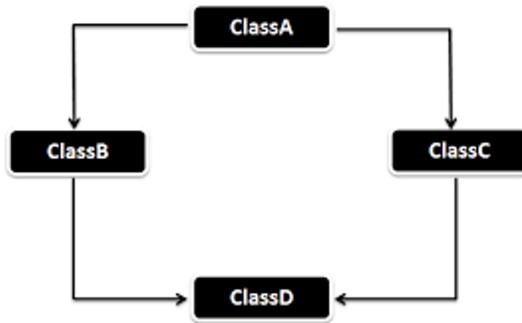
```
class D extends A {  
    public void print_D() { System.out.println("Class D"); }  
}
```

```
// Driver Class  
public class Test {  
    public static void main(String[] args)  
    {  
        B obj_B = new B();  
        obj_B.print_A();  
        obj_B.print_B();  
  
        C obj_C = new C();  
        obj_C.print_A();  
        obj_C.print_C();  
  
        D obj_D = new D();  
        obj_D.print_A();  
        obj_D.print_D();  
    }  
}
```

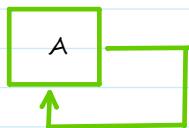
Output  
Class A  
Class B  
Class A  
Class C  
Class A  
Class D

#### 5. Hybrid Inheritance

Since Java does not support multiple inheritance, hybrid inheritance is also not possible in Java. Just like multiple inheritance you need to achieve this using interfaces.

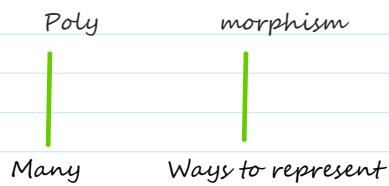


## 6. Cyclic Inheritance



A class cannot be own superclass. So this one also not supported.

## ➤ Polymorphism



Fact: Any language does not support Polymorphism that language does not consider as OOP.

- Types of Polymorphism

1. Compile time / Static polymorphism

Achieved via method overloading



Same name of the method but types, argument, return types, ordering can be different.

Example : Multiple constructor.

2. Runtime / Dynamic Polymorphism

Achieved by overriding

★ Parent obj = new child();

Able to access those function which are declared in reference class.

`classA objc = new classB();`



▼

But implementation always depends upon this class  
This is known as Upcasting

### Temp file

```
package com.OOP3.Inheritance;

public class Temp {
    void grett(){
        System.out.println("Hello world");
    }
    void abc(){
        System.out.println("abc");
    }
    void belong(){
        System.out.println("I belong to Temp class");
    }
}
```

### Temp2 file

```
package com.OOP3.Inheritance;

public class Temp2 extends Temp{
    @Override
    void grett(){
        System.out.println("Hello ladies & gentlemen");
    }
    void xyz(){
        System.out.println("xyz");
    }
    @Override
    void belong(){
        System.out.println("I belong to Temp2 class");
    }
}
```

### Main2 file

```
package com.OOP3.Inheritance;

public class Main2 {
    public static void main(String[] args) {
        Temp obj = new Temp2();
        obj.belong(); //I belong to Temp2 class
        obj.grett(); //Hello ladies & gentlemen
        obj.abc(); //abc //it is possible becoz temp2 class has extended temp class
    }
    // obj.xyz(); --> ERROR --> we can't access the function of Temp2 class
}

Temp2 obj2 = new Temp2();
obj2.xyz(); //xyz
obj2.belong(); //I belong to Temp2 class
/* It has overridden the Temp1 function */
```

### Using final with Inheritance:

The keyword final has three uses:

Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

Everything has to be same type, argument, return type.

Don't confuse with overloading

//remember overloading means same name of the method but types, argument, return type, ordering can be different.

How Java determine which method to run?  
Ans -> Dynamic method dispatched

# First, it can be used to create the equivalent of a named constant.

#### # Using final to Prevent Overriding:

To disallow a method from being overridden, specify final as a modifier at the start of its declaration.

Methods declared as final cannot be overridden.

Methods declared as final can sometimes provide a performance enhancement. The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with final methods.

Normally, Java resolves calls to methods dynamically, at run time. This is called late binding.

However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

#### # Using final to Prevent Inheritance:

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.

NOTE: Declaring a class as final implicitly declares all of its methods as final, too.

As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself & relies upon its subclasses to provide complete implementations.

# NOTE: Although static methods can be inherited, there is no point in overriding them in child classes because the method in parent class will run always no matter from which object you call it. That is why static interface methods cannot be inherited because these method will run from the parent interface and no matter if we were allowed to override them, they will always run the method in parent interface. That is why static interface method must have a body.

Static is independent. It does not depend upon its object

Overriding depends upon object

Static does not depend upon object

Hence static method can't be overridden

You can inherit but you cannot override

NOTE : Polymorphism does not apply to instance variables.

- Encapsulation

Wrapping up the implementation of the data members & the class.

- Abstraction

Hiding unnecessary detail & showing valuable information.

Let's have scenario in order to understand above this :

In order to start the car



You need the Key?

or

Do you need to know Internal Mechanics how the car starts



Obviously we need the car, all the unnecessary thing how the car works and all its important but we don't care.

See all this extra information hidden from us.

Obviously we need the car, all the unnecessary thing how the car works and all its important but we don't care.

See all this extra information hidden from us. This is known as **abstraction**.

```
Static void greet(){  
    System.out.println(".....");} → How it is printing we don't care.
```

```
ArrayList list = new ArrayList(); → How it is defining we don't care.
```

This is known as Abstraction data.

Whereas **encapsulation** is a method to hide the data in a single entity or unit along with a method to protect information from outside.

Story : We need sort type of methods.

How you do, what you add that's up to you.

All I need is whether the method is working or not.

So implementation of method that is known as **encapsulation**.

Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the <b>unnecessary</b> detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a <b>single</b> entity or unit so that the data can be protected from the outside world.
It solves an issue at the <b>design</b> level.	Encapsulation solves an issue at <b>implementation</b> level.
It focuses on the <b>external</b> lookout.	It focuses on <b>internal</b> working.
It can be implemented using <b>abstract classes</b> and <b>interfaces</b> .	It can be implemented by using the <b>access modifiers</b> (private, public, protected).
It is the process of <b>gaining</b> information.	It is the process of <b>containing</b> the information.
In abstraction, we use <b>abstract classes</b> and <b>interfaces</b> to hide the code complexities.	We use the <b>getters</b> and <b>setters</b> methods to hide the data.
The objects are <b>encapsulated</b> that helps to perform abstraction.	The object need not to <b>abstract</b> that result in encapsulation.

# Inheritance

19 July 2022 14:08

//Sample program to understand

## Box file

```
package com.OOP3.Inheritance;

package com.OOP3.Inheritance;

public class Box {
    // private double l; ---> other file cannot access those members of the superclass that have been declared as
    // private.
    //           ---> except this class no other class can access the private member
    double l;
    double w;
    double h;
    // double weight;

    //constructor overloading --> according to given argument the constructor has been selected.

    Box(double l, double w, double h) {
        this.l = l;
        this.w = w;
        this.h = h;
    }
    Box() {
        this.l = -1;
        this.w = -1;
        this.h = -1;
    }
    Box(double side) {
        this.l = side;
        this.w = side;
        this.h = side;
    }
    Box(Box obj) {
        this.l = obj.l;
        this.w = obj.w;
        this.h = obj.h;
    }

    public void calling(){
        System.out.println("-----");
    }
}
```

## BoxWeight file

```
package com.OOP3.Inheritance;

public class BoxWeight extends Box{
    double weight;
```

```

BoxWeight(double l, double w, double h, double weight) {
    super(l, w, h); //call the parent class constructor
        //used to initialise values present in parent class
    /*Java enforces that the call to super (explicit or not) must be the first statement in
     the constructor. This is to prevent the subclass part of the object being initialized
     prior to the superclass part of the object being initialized.*/
}

// super(); //default one will be called

System.out.println(this.l);
System.out.println(super.l);

/*
--> you might have question in your mind that if we are able to take value from this.l then
    why there is need of super.l
--> ans is : super keyword in simple terms refers to superclass (parent) objects
--> let's have one scenario :-
System.out.println(super.weight); here with the help of super keyword we are able to refer the
superclass. super has having no idea what base class contain.
System.out.println(this.weight); here we are taking the value from this class only

*/
this.weight = weight;
}

```

```

BoxWeight(Box other, double weight) {
    super(other);
    this.weight = weight;
}

BoxWeight(double weight) {
    super();
    this.weight = weight;
}

BoxWeight() {
    super();
    this.weight = -1;
}

```

### //Boxprice file

```

package com.OOP3.Inheritance;

//Remember : constructor is required in a parent class for inheriting

public class BoxPrice extends BoxWeight{
    double cost;
}

```

```

public BoxPrice(double l, double w, double h, double weight, double cost) {
    super(l, w, h, weight);
    this.cost = cost;
}

public BoxPrice(Box other, double weight, double cost) {
    super(other, weight);
    this.cost = cost;
}

public BoxPrice(double weight, double cost) {
    super(weight);
    this.cost = cost;
}

```

### //Main file

```
package com.OOP3.Inheritance;
```

```

public class Main {
    public static void main(String[] args) {
        Box box1 = new Box(7,8,9);
        System.out.println(box1.l + " " + box1.w + " " + box1.h);

        Box box2 = new Box(box1);
        System.out.println(box2.l + " " + box2.w + " " + box2.h);

        BoxWeight box3 = new BoxWeight(4,3,2,1);
        System.out.println(box3.l + " " + box3.w + " " + box3.h + " " + box3.weight);

        Box box5 = new BoxWeight(2,3,4,8);
        System.out.println(box5.w);
//        System.out.println(box5.weight); -->error

        //there are many variables in both parent and child classes
        //you are given access to variables that are in the ref type i.e, BoxWeight
        //hence, you should have access to weight variable
        //this also means, that the ones you are trying to access should be initialised
        //but here, when the obj itself is of type parent class, how will you call the constructor of child class
        //this is why error
//        BoxWeight box6 = new Box(2,3,4);
//        System.out.println(box6.weight);

        BoxPrice box8 = new BoxPrice(2,3,4,5,6);
        System.out.println(box8.w);

        BoxPrice box9 = new BoxPrice(box1,24,26);
        System.out.println(box9.weight);
    }
}
```



# Polymorphism

21 July 2022 03:48

//shape file

```
package com.OOP3.Polymopherism;

public class Shape {
    void area(){
        System.out.println("-----");
    }

    /*
        final : final is used to prevent overriding
        final void area(){
            System.out.println("-----");
        }
    */

    static void greetings(){
        System.out.println("Hello Buddy, I am from shape class ");
    }
}
```

//circle file

```
package com.OOP3.Polymopherism;

public class Circle extends Shape{
    //this will run when obj of circle is created
    //hence it is overriding the parent method
    @Override //this is called annotation //use for check purposes
    void area(){
        System.out.println("pi * r * r");
    }
}

package com.OOP3.Polymopherism;

public class Circle extends Shape{
```

```
//this will run when obj of circle is created  
//hence it is overriding the parent method  
@Override //this is called annotation //use for check purposes  
void area(){  
    System.out.println("pi * r * r");  
}  
}
```

### //square file

```
package com.OOP3.Polymorphism;
```

```
public class Square extends Shape{  
    void area(){  
        System.out.println("a*a");  
    }  
    /*
```

@Override --> Overriding depends upon object  
static does not depends upon object  
Hence static method can't be overridden.

```
*/  
// static void greetings(){  
//     System.out.println("Hello Buddy, I am from square class ");  
// }  
}
```

## Access Control, In-built Packages, Object Class

11 July 2022 04:18

How a member can be accessed is determined by the access modifier attached to its declaration. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

Java's access modifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved.

When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

	class	package	subclass (same pkg)	subclass (diff pkg)	world (diff pkg & not subclass)
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y	
no modifier	Y	Y	Y		
private	Y				

Y : accessible

blank : not accessible

```
package packageOne;
public class Base
{
    protected void display(){
        System.out.println("in Base");
    }
}

package packageTwo;
public class Derived extends packageOne.Base{
    public void show(){
        new Base().display(); // this is not working
        new Derived().display(); // is working
        display(); // is working
    }
}
```

protected allows access from subclasses and from other classes in the same package.  
We can use child class to use protected member outside the package but only child class object can

access it.

That's why any Derived class instance can access the protected method in Base.

The other line creates a Base instance (not a Derived instance!!).

And access to protected methods of that instance is only allowed from objects of the same package.

new Derived().display();

-> allowed, because the caller, an instance of Derived has access to protected members and fields of its subclasses, even if they're in different packages

new Derived().display();

-> allowed, because you call the method on an instance of Derived and that instance has access to the protected methods of its subclasses

new Base().display();

-> not allowed because the caller's (the this instance) class is not defined in the same package like the Base class, so this can't access the protected method. And it doesn't matter - as we see - that the current subclasses a class from that package. That backdoor is closed ;)

Remember that any time talks about a subclass having an access to a superclass member, we could be talking about the subclass inheriting the member, not simple accessing the member through a reference to an instance of the superclass.

class C

protected member; // in a different package

class S extends C

obj.member; // only allowed if type of obj is S or subclass of S

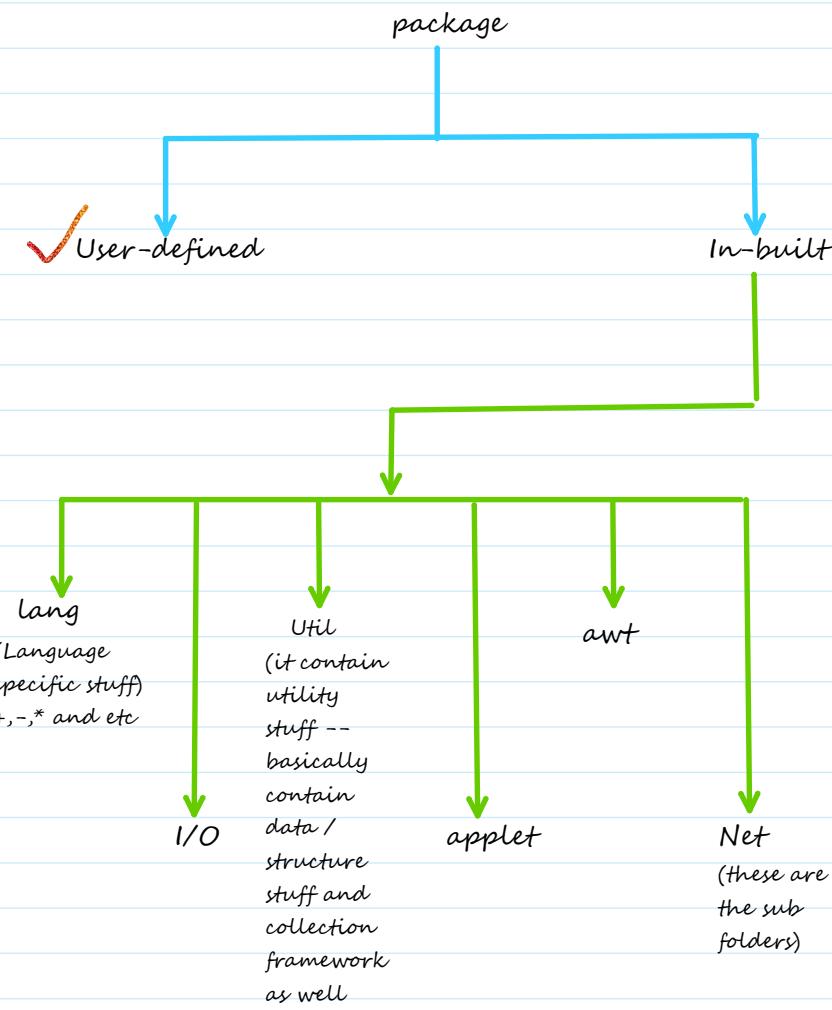
The motivation is probably as following. If obj is an S, class S has sufficient knowledge of its internals, it has the right to manipulate its members, and it can do this safely. If obj is not an S, it's probably another subclass S2 of C, which S has no idea of. S2 may have not even been born when S is written. For S to manipulate S2's protected internals is quite dangerous.

If this is allowed, from S2's point of view, it doesn't know who will tamper with its protected internals and how, this makes S2 job very hard to reason about its own state.

Now if obj is D, and D extends S, is it dangerous for S to access obj.member? Not really.

How S uses member is a shared knowledge of S and all its subclasses, including D. S as the superclass has the right to define behaviours, and D as the subclass has the obligation to accept and conform.

For easier understanding, the rule should really be simplified to require obj's (static) type to be exactly S. After all, it's very unusual and inappropriate for subclass D to appear in S. And even if it happens, that the static type of obj is D, our simplified rule can deal with it easily by upcasting: ((S)obj).member



## Abstract Classes, Interfaces, Annotations

12 July 2022 01:10

Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.

In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.

You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.

```
abstract type name(parameter-list);
```

These methods are sometimes referred to as subclass's responsibility because they have no implementation specified in the superclass.

Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

Any class that contains one or more abstract methods must also be declared abstract.

# There can be no objects of an abstract class.

# You cannot declare abstract constructors, or abstract static methods.

```
abstract static void career(); //X
```

# You can declare static methods in abstract class.

```
public abstract class Parent {  
    abstract void career();  
    abstract void partner();  
  
    static void hello(){  
        System.out.println("hello");  
    }  
  
    void normal(){  
        System.out.println("Normal method");  
    }  
}
```

In main file

Parent.normal ---> you can call it -->"Normal method";

Or in child class file you can override as well

```
@override  
void normal(){  
    super.normal();  
}
```

Because there can be no objects for abstract class. If they had allowed to call abstract static methods, it would mean we are calling an empty method (abstract) through classname because it is static.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

Abstract classes can include as much implementation as they see fit i.e. there can be concrete methods(methods with body) in abstract class.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

A public constructor on an abstract class doesn't make any sense because you can't instantiate an abstract class directly (can only instantiate through a derived type that itself is not marked as abstract).

Check: <https://stackoverflow.com/questions/260666/can-an-abstract-class-have-a-constructor>

Abstract class vs Interface:

Type of methods:

Interface can have only abstract methods.

Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.

Final Variables:

Variables declared in a Java interface are by default final.

An abstract class may contain non-final variables.

Type of variables:

Abstract class can have final, non-final, static and non-static variables.

Interface has only static and final variables.

Implementation:

Abstract class can provide the implementation of interface.

Interface can't provide the implementation of abstract class.

Inheritance vs Abstraction:

A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".

Multiple implementation:

An interface can extend another Java interface only,

an abstract class can extend another Java class and implement multiple Java interfaces.

Accessibility of Data Members:

Members of a Java interface are public by default.

A Java abstract class can have class members like private, protected, etc.

# Generics, Custom ArrayList

23 July 2022 13:39

## • Custom ArrayList

In this post i will be explaining *ArrayList Custom* implementation.

Initially, when we declare `ArrayList<Integer>` with `INITIAL_CAPACITY = 10`, it will be like this-

0	1	2	3	4	5	6	7	8	9
null									

Let's add(71) in ArrayList, after addition our ArrayList will look like this-

0	1	2	3	4	5	6	7	8	9
71	null								

Suppose we have filled all the 10 box then more 10 box will be added. This is how ArrayList capacity works.

```
package com.OOP6;
```

```
import java.util.ArrayList;
import java.util.Arrays;

public class CustomArrayList {

    private int[] data;
    private static int DEFAULT_SIZE = 10;
    private int size = 0; // also working as index value

    public CustomArrayList() {
        this.data = new int[DEFAULT_SIZE];
    }

    public void add(int num) {
        if (isFull()) {
            resize();
        }
        data[size++] = num;
    }

    private void resize() {
        int[] temp = new int[data.length * 2];

        // copy the current items in the new array
        for (int i = 0; i < data.length; i++) {
```

```

        temp[i] = data[i];
    }
    data = temp;
}

private boolean isFull() {
    return size == data.length;
}

// it will delete last index
public int remove() {
    int removed = data[--size];
    return removed;
}

public int get(int index) {
    return data[index];
}

public int size() {
    return size;
}

public void set(int index, int value) {
    data[index] = value;
}

@Override
public String toString() {
    return "CustomArrayList{" +
        "data=" + Arrays.toString(data) +
        ", size=" + size +
        '}';
}

public static void main(String[] args) {
    // ArrayList list = new ArrayList();
    CustomArrayList list = new CustomArrayList();
    // list.add(3);
    // list.add(5);
    // list.add(9);

    for (int i = 0; i < 14; i++) {
        list.add(2 * i);
    }
    list.remove();

    System.out.println(list);

    ArrayList<Integer> list2 = new ArrayList<>(); // in this list only integer type can be stored & <Integer> : known as generic
    // list2.add("dfghj");
}
}

• Custom generic Arraylist

```

**Type safety**

Suppose we want that client should enter number. If we don't put generic it will allow the Integer input as well other data type also (which we don't want).

```
import java.util.ArrayList;
import java.util.Arrays;

// https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects

public class CustomGenArrayList<T> {

    private Object[] data;
    private static int DEFAULT_SIZE = 10;
    private int size = 0; // also working as index value

    public CustomGenArrayList() {
        data = new Object[DEFAULT_SIZE];
    }

    public void add(T num) {
        if (isFull()) {
            resize();
        }
        data[size++] = num;
    }

    private void resize() {
        Object[] temp = new Object[data.length * 2];

        // copy the current items in the new array
        for (int i = 0; i < data.length; i++) {
            temp[i] = data[i];
        }
        data = temp;
    }

    private boolean isFull() {
        return size == data.length;
    }

    public T remove() {
        T removed = (T)(data[--size]);
        return removed;
    }

    public T get(int index) {
        return (T)data[index];
    }

    public int size() {
        return size;
    }

    public void set(int index, T value) {
        data[index] = value;
    }

    @Override
    public String toString() {
        return "CustomGenArrayList{" +
            "data=" + Arrays.toString(data) +
    }
```

```

    ", size=" + size +
    '}';
}

public static void main(String[] args) {
//    ArrayList list = new ArrayList();
    CustomGenArrayList list = new CustomGenArrayList();
//    list.add(3);
//    list.add(5);
//    list.add(9);

//    for (int i = 0; i < 14; i++) {
//        list.add(2 * i);
//    }

//    System.out.println(list);

ArrayList<Integer> list2 = new ArrayList<>();
//    list2.add("dfghj");

CustomGenArrayList<Integer> list3 = new CustomGenArrayList<>();
for (int i = 0; i < 14; i++) {
    list3.add(2 * i);
}

System.out.println(list3);

}
}

```

## ? • Java Wildcard

```

package com.OOP6;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

// https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects

// here T should either be Number or its subclasses
public class WildcardExample<T extends Number> {

    private Object[] data;
    private static int DEFAULT_SIZE = 10;
    private int size = 0; // also working as inde value

    public WildcardExample() {
        data = new Object[DEFAULT_SIZE];
    }

    public void getList(List<? extends Number> list) {
        // do something
    }

    public void add(T num) {
        if (isFull()) {

```

```
    resize();
}
data[size++] = num;
}

private void resize() {
    Object[] temp = new Object[data.length * 2];

    // copy the current items in the new array
    for (int i = 0; i < data.length; i++) {
        temp[i] = data[i];
    }
    data = temp;
}

private boolean isFull() {
    return size == data.length;
}

public T remove() {
    T removed = (T)(data[--size]);
    return removed;
}

public T get(int index) {
    return (T)data[index];
}

public int size() {
    return size;
}

public void set(int index, T value) {
    data[index] = value;
}

@Override
public String toString() {
    return "CustomGenArrayList{" +
        "data=" + Arrays.toString(data) +
        ", size=" + size +
        '}';
}

public static void main(String[] args) {
//    ArrayList list = new ArrayList();
    WildcardExample list = new WildcardExample();
//    list.add(3);
//    list.add(5);
//    list.add(9);

//    for (int i = 0; i < 14; i++) {
//        list.add(2 * i);
//    }

//    System.out.println(list);

    ArrayList<Integer> list2 = new ArrayList<>();
```

```
//      list2.add("dfghj");

WildcardExample<Integer> list3 = new WildcardExample<>();
for (int i = 0; i < 14; i++) {
    list3.add(2 * i);
}

System.out.println(list3);

}
```

# Compression operator , Lambda Expressions, Exception Handling, Object Cloning

08 August 2022 01:21

## #student file

```
package com.OOP6.comparision;

public class Student implements Comparable<Student>{
    int rollno;
    float marks;

    public Student(int rollno, float marks) {
        this.rollno = rollno;
        this.marks = marks;
    }

    @Override
    public int compareTo(Student o) {
        // System.out.println("in compareto method");
        int diff = (int)(this.marks - o.marks);

        // if diff == 0: means both are equal
        // if diff < 0: means o is bigger else o is smaller

        return diff;
    }

    @Override
    public String toString() {
        return marks + "";
    }
}
```

## #main file

```
package com.OOP6.comparision;

import java.util.Arrays;

import static java.util.Arrays.sort;

public class Main{
    public static void main(String[] args) {
        Student rahul = new Student(59,69.36f);
        Student raj = new Student(5,96.36f);
        Student piyush = new Student(51,90.36f);
        Student karan = new Student(25,86.36f);
        Student sanjay = new Student(10,59.36f);

        //here compiler will be confused bcz here 2 arguments are there.
        //to solve this problem we have to implement comparable in class file.
```

```
// so to compare the value this is not the convenient way.  
// if(raj.marks>rahul.marks){  
//     System.out.println("raj has more marks");  
// }  
  
Student[] list = {rahul, raj, piyush, karan, sanjay};  
System.out.println(Arrays.toString(list));  
Arrays.sort(list);  
System.out.println(Arrays.toString(list));  
  
// if(raj.compareTo(rahul)>0){  
//     System.out.println("raj has more marks");  
// }  
}  
}
```

### ➤ Lambda function

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value.

Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

#### Syntax

The simplest lambda expression contains a single parameter and an expression:

parameter -> expression

To use more than one parameter, wrap them in parentheses:

(parameter1, parameter2) -> expression

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

(parameter1, parameter2) -> {code block }

# OOP6

24 July 2022 01:03

## Points to remember

04 September 2022 18:16

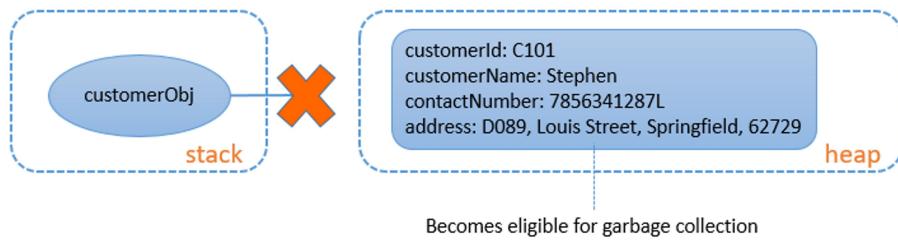
- If the class is public it doesn't mean that the data type and function inside the class that also public.

# Memory management

04 February 2023 03:48

## Objects eligible for garbage collection

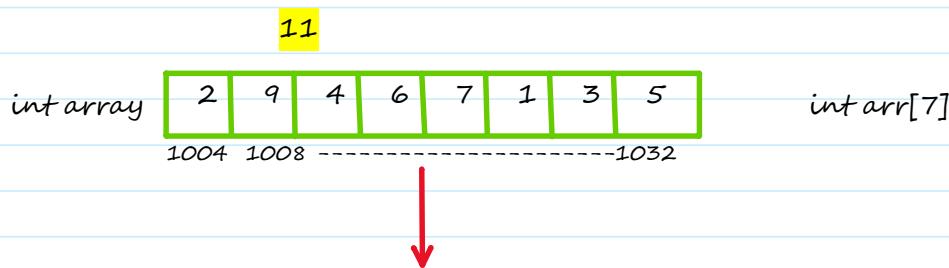
When the reference variable pointing to the object is initialized to null, the object will not have any reference.



```
public class Tester {  
    public static void main(String[] args) {  
        // Object creation  
        Customer customerObj = new Customer("C101", "Stephen Abram",  
                                           7856341287L, "D089, St. Louis Street, Springfield, 62729");  
        // Reference variable initialized to null  
        customerObj = null;  
    }  
}
```

## LINKED LIST

13 August 2022 12:20



- Suppose after sometime we want to add 11 after 9 in this array then how you will do?
- For that first we will increase the length of the array which is not possible while writing the code. we cannot change the length of the array during runtime.
- For increasing the length of array we have to **re-declare** the array again.
- Lets take we re-declared then also for adding the 11 after 9 we have to do many shifts of next values.

Think like 11 if we want to add many values then program will become so complex and we have to redeclare the length of array and shifting of the value again and again.

To counter such problem linked list comes into the play

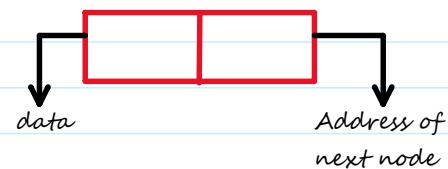
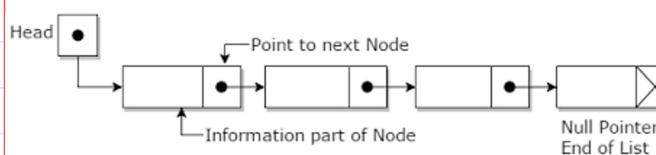
2	9	11	4	6	7	1	3	5
1004	1008	-----	-----	-----	1032			

- **Linked list**—A linked list consist of nodes where each node contains a data field and a reference(link) to the next node in the list.



Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a **linear data structure**.

It is linear because the data can be inserted or removed only in one dimension—lengthwise, making the list either longer or shorter. The elements are called 'nodes' and consist of two parts : data and link (address of next node). It should be viewed as



- Here a pointer stores the address of the first node or element of the list, and that element points to the next node and so on. The last element doesn't point to any address indicating the

end of the list.

2. While adding or removing a node other elements are not required to be shifted. Only the pointer of last node is needed to be changed to the address of new node while addition. And only change in the link portion of previous node has to be done while deleting a node.
3. We only allocate memory for the nodes we require, this causes an efficient memory utilisation.

- **Advantage -**

- No size limitation

- Insertion / deletion easy

- **Drawbacks of Linked Lists:**

1. Accessing a node is time taking, as it is needed to be started from the head and go up to the required node.
2. Going reverse in a linked list is not possible. This is rectified in a doubly linked list, where pointer to both previous and next nodes are stored with data in each node.
3. Extra memory space for pointers is required (for every node, extra space for a pointer is needed)
4. Random access is not allowed as elements are not stored in contiguous memory locations.

- Application- browsing web pages, going previous and next

- now we can insert any values easily wherever we want by the help of linked list

➤ **Operation in linked list**

- Traversal : access each element of the linked list.
- Insertion : adds a new element to the linked list.
- Deletion ; remove the existing elements.
- Search : find a node in the linked list
- Sort : sort the node of the linked list.

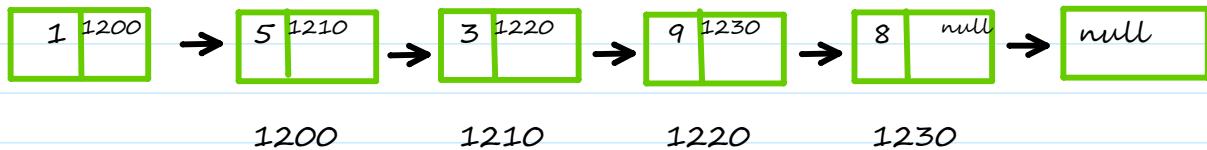
# Singly linked list

24 August 2022 00:06

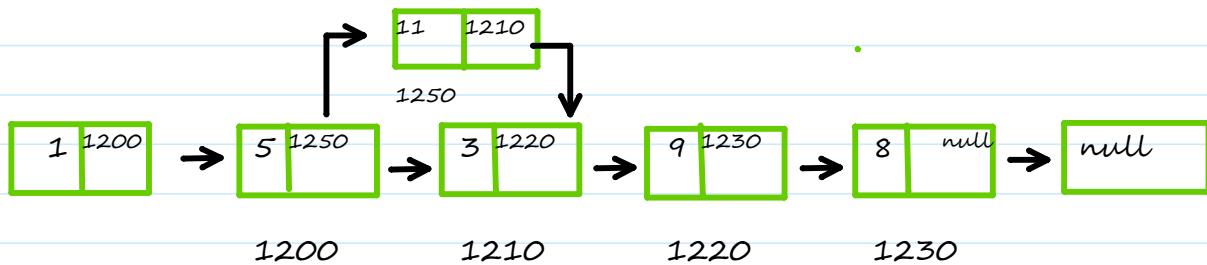
Singly linked lists contain nodes which have a data field as well as a next field, which points to the next node in the sequence.

~ Suppose here we have to add 11 after 5 now we can do easily

Before -->



After -->



Program:

```
package com.LinkedList;
package com.LinkedList;

import com.LinkedList.LLquestion.Rec_LL;

public class LI {
    class Node{
        int value;
        Node next;

        public Node(int value) {
            this.value = value;
            this.next=null;
        }
        public Node(int value,Node next) {
            this.value = value;
            this.next=next;
        }
    }
    public Node head = null;
    public Node tail = null;
    public int size=0;
    public void insertFirst(int val){
```

```
Node newNode = new Node(val); //one node created

if(head == null) {
    head = newNode;
    tail = newNode;
}
else {
    newNode.next=head;
    head=newNode;
}
size++;

}

public void insertLast(int val){
    if (tail == null) {
        insertFirst(val);
        return;
    }
    Node node = new Node(val);
    tail.next = node;
    tail = node;
    size++;
}

public void insertMid(int value,int pos){
    if(pos==size) {
        insertLast(value);
        return;
    }
    if(pos==0) {
        insertFirst(value);
        return;
    }

    Node tempNode=head;

    for (int i = 1; i < pos; i++) {
        tempNode=tempNode.next;
    }
    Node newNode = new Node(value,tempNode.next);
    tempNode.next=newNode;

    size++;

}

public void deleteFirst(){
    head=head.next;
    if(head==null){
        tail=null;
    }
}
```

```
size--;
return;
}
public void deleteLast(){
if(size<=1) {
    deleteFirst();
    return;
}
Node secondLast=get(size-2);
secondLast.next=null;
size--;
}
public void deleteMid(int index){
    Node node = get(index-1);
    Node temp = node.next;
    node.next=temp.next;
    size--;
}

}
public Node get(int index){
    Node node = head;
    for (int i = 0; i < index; i++) {
        node=node.next;
    }
    return node;
}

}

public void display(){
    Node current=head;
    if(head == null) System.out.println("List is empty");
    System.out.println("Nodes of singly LinkedList :");
    while (current != null){
        System.out.print(current.value+" ");
        current=current.next;
    }
    System.out.println();
}

}

public static void main(String[] args) {
    LL linkedlist = new LL();
    linkedlist.insertFirst(25);
    linkedlist.insertFirst(26);
    linkedlist.insertFirst(27);
    linkedlist.insertFirst(28);
    linkedlist.insertFirst(29);
    linkedlist.insertLast(111);
    linkedlist.insertFirst(44 );
    //    linkedlist.insertLast(31);
    //    linkedlist.insertMid(66,3);
    //    linkedlist.deleteMid(3);
}
```

```
linkedlist.display();  
System.out.println(linkedlist.size());  
  
}  
}
```

# Sorting in LinkedList

16 September 2022 02:24

We can sort the LinkedList by many sorting techniques:

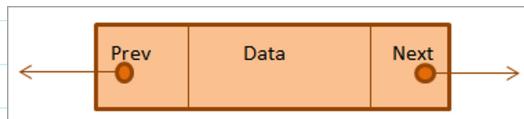
1. [Bubble sort](#)
2. [Insertion sort](#)
3. [Quick sort](#)
4. [Merge sort](#)

## Doubly Linked List

15 August 2022 11:03

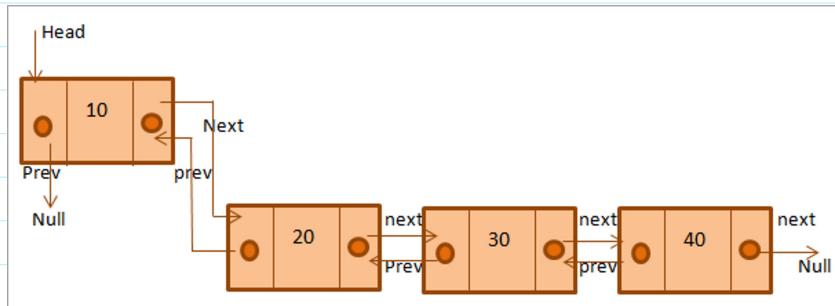
A linked list has another variation called "doubly linked list". A doubly linked list has an additional pointer known as the previous pointer in its node apart from the data part and the next pointer as in the singly linked list.

A node in the doubly linked list looks as follows:



Here, "Prev" and "Next" are pointers to the previous and next elements of the node respectively. The 'Data' is the actual element that is stored in the node.

The following figure shows a doubly linked list.



The above diagram shows the doubly linked list. There are four nodes in this list. As you can see, the previous pointer of the first node, and the next pointer of the last node is set to null. The previous pointer set to null indicates that this is the first node in the doubly linked list while the next pointer set to null indicates the node is the last node.

**Advantage :** As each node has pointers pointing to the previous and next nodes, the doubly linked list can be traversed easily in forward as well as backward direction

You can quickly add the new node just by changing the pointers.

Similarly, for delete operation since we have previous as well as next pointers, the deletion is easier and we need not traverse the whole list to find the previous node as in case of the singly linked list.

**Disadvantages :** Since there is an extra pointer in the doubly linked list i.e. the previous pointer, additional memory space is required to store this pointer along with the next pointer and data item.

All the operations like addition, deletion, etc. require that both previous and next pointers are manipulated thus imposing operational overhead.

# Doubly linked list code

20 August 2022 01:04

*Deletion, sorting has to be done*

```
package com.LinkedList;

public class Dll {
    class Node {
        int value;
        Node next;
        Node prev;

        public Node(int value) {
            this.value = value;
        }
    }

    public int size=0;
    public Node head=null;
    public Node tail=null;

    public void insertFirst(int value){
        Node newNode = new Node(value);

        if(head == null) {
            head = newNode;
            tail = newNode;
        }
        else {
            newNode.next=head;
            newNode.prev=null;
            head.prev=newNode;
            head=newNode;
        }
        size++;
    }

    public void insertMid(int pos, int value){
        if(pos==0) {
            insertFirst(value);
        }
        else {
            Node current = head;
            for(int i=1; i<pos; i++) {
                current = current.next;
            }
            Node newNode = new Node(value);
            newNode.next=current.next;
            newNode.prev=current;
            current.next.prev=newNode;
            current.next=newNode;
        }
    }
}
```

```
        return;
    }
    if(pos==size) {
        insertLast(value);
        return;
    }
    Node newNode = new Node(value);
    Node currentNode = head;
    for (int i = 0; i < pos-2; i++) {
        currentNode=currentNode.next;
    }
    newNode.next=currentNode.next;
    newNode.prev=currentNode;
    currentNode.next=newNode;

    //going two node ahead to give previous node location
    currentNode=currentNode.next.next;
    currentNode.prev=newNode;

    size++;
}
public void insertLast(int value){
    Node newNode = new Node(value);
    Node currentNode = tail;
    currentNode.next=newNode;
    newNode.prev=currentNode;
    tail=newNode;

    size++;
}

public void display(){
    Node current=head;
    if(head == null) System.out.println("List is empty");
    System.out.println("Nodes of singly LinkedList : ");
    while (current != null){
        System.out.print(current.value+" -> ");
        current=current.next;
    }
}
```

```
System.out.println("END");

}

public void displayReversal(){
    Node currentNode = tail;
    if(head == null) System.out.println("List is empty");
    System.out.println("Nodes of Doubly LinkedList : ");
    for (int i = size; i > 0 ; i--) {
        System.out.print(currentNode.value+ " -> ");
        currentNode=currentNode.prev;
    }
    System.out.println();
}

public static void main(String[] args) {
    Dll doubly = new Dll();
    doubly.insertFirst(24);
    doubly.insertFirst(78);
    doubly.insertFirst(88);
    doubly.insertFirst(100);
    doubly.insertLast(1);
    doubly.display();
    doubly.insertMid(2,177);

    doubly.display();
    doubly.displayReversal();
}

}
```

# Circular Linked List

20 August 2022 00:38

## Circular Linked List

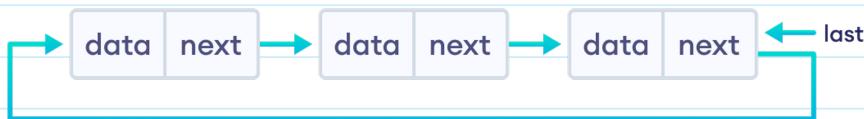
In this article, you will learn what circular linked list is and its types with implementation.

A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle.

There are basically two types of circular linked list.

### 1. Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.



### 2. Circular Doubly Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



# Singly circular linked list code

20 August 2022 00:51

sorting

## Program

```
package com.LinkedList;

public class CircularLL {

    private Node head;
    private Node tail;

    public CircularLL() {
        this.head = null;
        this.tail = null;
    }

    public void insert(int val) {
        Node node = new Node(val);
        if (head == null) {
            head = node;
            tail = node;
            return;
        }

        tail.next = node;
        node.next = head;
        tail = node;
    }

    public void display() {
        Node node = head;
        if (head != null) {
            do {
                System.out.print(node.val + " -> ");
                if (node.next != null) {
                    node = node.next;
                }
            } while (node != head);
        }
    }
}
```

```
        }
        System.out.println("HEAD");
    }

    public void delete(int val) {
        Node node = head;
        if (node == null) {
            return;
        }

        if (head == tail){
            head = null;
            tail = null;
            return;
        }

        if (node.val == val) {
            head = head.next;
            tail.next = head;
            return;
        }

        do {
            Node n = node.next;
            if (n.val == val) {
                node.next = n.next;
                break;
            }
            node = node.next;
        } while (node != head);

    }

    private class Node {
        int val;
        Node next;

        public Node(int val) {
            this.val = val;
        }
    }
}
```

```
}
```

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

```
    CircularLL cll = new CircularLL();
```

```
    cll.insert(0);
```

```
    cll.insert(100);
```

```
    cll.insert(200);
```

```
    cll.insert(300);
```

```
    cll.insert(400);
```

```
    cll.display();
```

```
    cll.delete(200);
```

```
    cll.display();
```

```
}
```

```
}
```

# Doubly circular linked list code

20 August 2022 00:51

sorting

```
class GFG {  
  
    // structure of a Node  
    static class Node {  
        int data;  
        Node next;  
        Node prev;  
    };  
  
    // Function to insert node in the list  
    static Node insert(Node start, int value)  
    {  
        // If the list is empty, create a single node  
        // circular and doubly list  
        if (start == null) {  
            Node new_node = new Node();  
            new_node.data = value;  
            new_node.next = new_node.prev = new_node;  
            start = new_node;  
            return start;  
        }  
  
        // If list is not empty  
  
        // Find last node /  
        Node last = (start).prev;  
  
        // Create Node dynamically  
        Node new_node = new Node();  
        new_node.data = value;  
  
        // Start is going to be next of new_node  
        new_node.next = start;  
  
        // Make new node previous of start  
        (start).prev = new_node;  
  
        // Make last previous of new node
```

```
new_node.prev = last;

// Make new node next of old last
last.next = new_node;
return start;
}

// Function to delete a given node from the list
static Node deleteNode(Node start, int key)
{
    // If list is empty
    if (start == null)
        return null;

    // Find the required node
    // Declare two pointers and initialize them
    Node curr = start, prev_1 = null;
    while (curr.data != key) {
        // If node is not present in the list
        if (curr.next == start) {
            System.out.printf("\nList doesn't have node with value = %d", key);
            return start;
        }

        prev_1 = curr;
        curr = curr.next;
    }

    // Check if node is the only node in list
    if (curr.next == start && prev_1 == null) {
        (start) = null;
        return start;
    }

    // If list has more than one node,
    // check if it is the first node
    if (curr == start) {
        // Move prev_1 to last node
        prev_1 = (start).prev;

        // Move start ahead
        start = (start).next;
    }
}
```

```

        // Adjust the pointers of prev_1 and start node
        prev_1.next = start;
        (start).prev = prev_1;
    }

    // check if it is the last node
    else if (curr.next == start) {
        // Adjust the pointers of prev_1 and start node
        prev_1.next = start;
        (start).prev = prev_1;
    }
    else {
        // create new pointer, points to next of curr node
        Node temp = curr.next;

        // Adjust the pointers of prev_1 and temp node
        prev_1.next = temp;
        temp.prev = prev_1;
    }
    return start;
}

// Function to display list elements
static void display(Node start)
{
    Node temp = start;

    while (temp.next != start) {
        System.out.printf("%d ", temp.data);
        temp = temp.next;
    }
    System.out.printf("%d ", temp.data);
}

// Driver program to test above functions
public static void main(String args[])
{
    // Start with the empty list
    Node start = null;

    // Created linked list will be 4.5.6.7.8
}

```

```
start = insert(start, 4);
start = insert(start, 5);
start = insert(start, 6);
start = insert(start, 7);
start = insert(start, 8);

System.out.printf("List Before Deletion: ");
display(start);

// Delete the node which is not present in list
start = deleteNode(start, 9);
System.out.printf("\nList After Deletion: ");
display(start);

// Delete the first node
start = deleteNode(start, 4);
System.out.printf("\nList After Deleting %d: ", 4);
display(start);

// Delete the last node
start = deleteNode(start, 8);
System.out.printf("\nList After Deleting %d: ", 8);
display(start);

// Delete the middle node
start = deleteNode(start, 6);
System.out.printf("\nList After Deleting %d: ", 6);
display(start);

}
```

# Question tips

01 October 2022 19:37

- Whenever you find question on linked list cycle go for fast and slow pointer process
  - Move first pointer by 2 & slow pointer by 1.
  - When first slow points same node it means the given linked list is cycle linked list or else it would h point null
  - Time complexity
    - If cycle is there  $O(n)$ .
    - If cycle is not there  $O(n)$ .

# Some basic points

20 August 2022 01:10

- To go middle of the linked list

Take two reference one at first node and second at second node

Move first reference by one position & move 2nd reference by 2 position and you have to do like this till conditions get true.

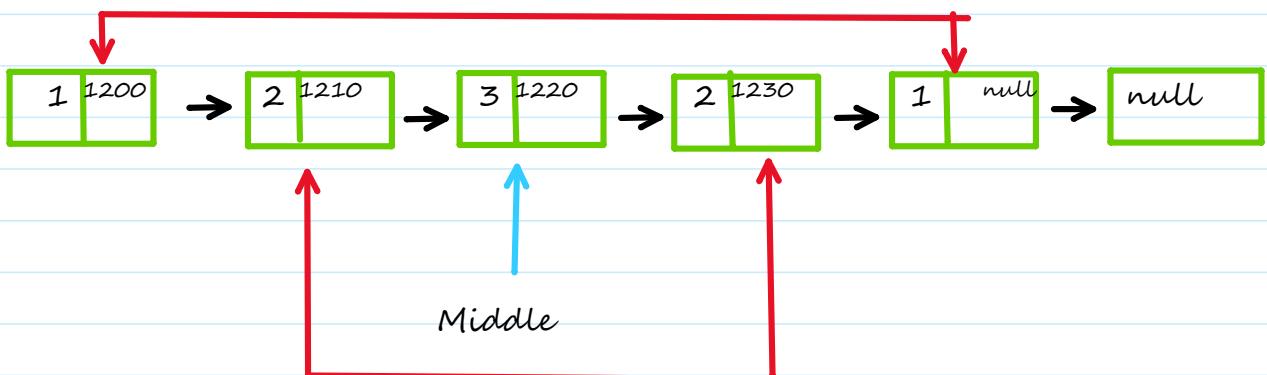
Condition: if(secondRef==null || secondRef.next==null)



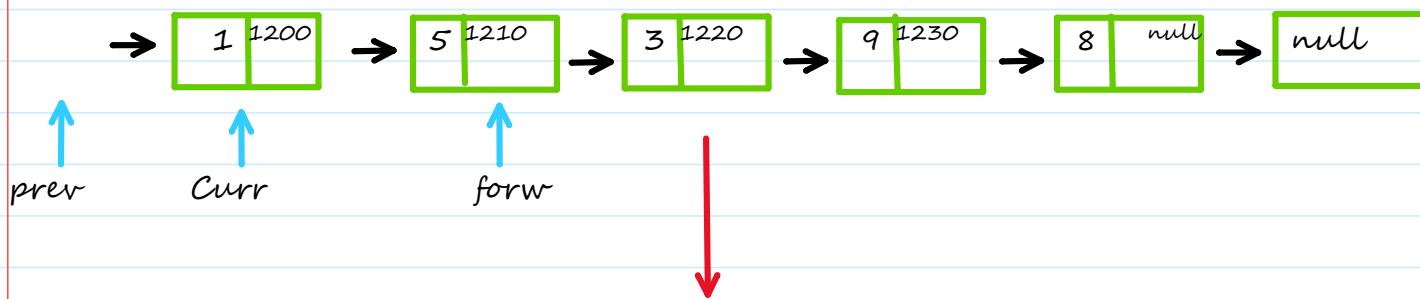
- Palindrome

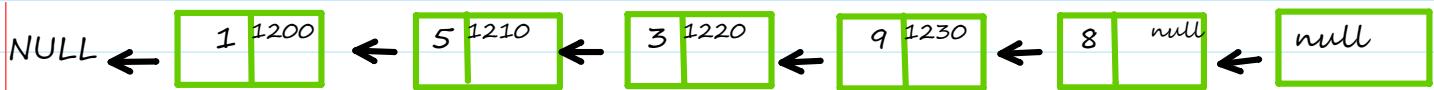
-> go to the middle node

-> start checking (head to middle) with (end to middle)



- Reverse of singly linked list





`curr.next = prev`

`prev = curr`

`curr = forw`

`forw = curr.next`

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow$

Suppose you want to access the value of node 2 by staying in node 1 than simply you have to do is : `node.next.value`;

- To check whether next node is null or not (especially in while loop) then go for `node != null`

`node.next != null` (Wrong way bcz here we are not going till last node)

# Stack

20 August 2022 02:31

- Last in first out



## Methods in Stack Class

**empty()** It returns true if nothing is on the top of the stack. Else, returns false.

**peek()** Returns the element on the top of the stack, but does not remove it.

**pop()** Removes and returns the top element of the stack. An 'EmptyStackException'

An exception is thrown if we call pop() when the invoking stack is empty.

**push(Object element)** Pushes an element on the top of the stack.

**search(Object element)** It determines whether an object exists in the stack. If the element is found, It returns the position of the element from the top of the stack. Else, it returns -1.

# Queue

21 August 2022 00:22

- A queue is an object that represents a data structure designed to have the element inserted at the end of the queue, and the element removed from the beginning of the queue.
- First in first out



Remember queue like cinema:  
People come to purchase ticket  
those whoe will come first will  
get ticket first  
//FIFO

```
package com.stack_queue;

import java.util.LinkedList;
import java.util.Queue;

public class Q {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        queue.add(3);
        queue.add(4);
        queue.add(5);

        System.out.println(queue.peek()); //1
        System.out.println(queue.remove()); //1
        System.out.println(queue.remove()); //2
    }
}
```

- Java deque : Basically you can insert & remove from the both of the side
  - ◊ You can insert or remove from front.
  - ◊ You can insert or remove from back.
  - ◊ Null element are not allowed in this.
  - ◊ It doesn't has size restriction.
  - ◊ It's than linked list and stack.

Program :

```
package com.stack_queue;

import java.util.ArrayDeque;
import java.util.Deque;

public class DQ {
    public static void main(String[] args) {
        Deque<Integer> deque = new ArrayDeque<>();
        deque.add(25);
        deque.add(96);
        deque.addLast(55);
```

```
deque.addFirst(44);
deque.add(111);
//25
//25 96
//25 96 55
//44 25 96 55
//44 25 96 55 111
System.out.println(deque.removeFirst()); //44
System.out.println(deque.removeLast()); //111
System.out.println(deque.remove()); //25
}
}
```

# ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	=	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	:	91	5B	133	{	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135	}	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

# HASHMAP

18 June 2022 02:58

## ➤ What is a Hashmap?

A hashmap is a data structure which has an amazing property that most of the operations which we perform on it are done in  $O(1)$  time complexity. The data is stored in a hash-map in the form of key-value pairs. For instance: consider the following data about the population of various countries:

Country	Population (in millions)
India	1391
China	1398
USA	329
Indonesia	268

If the data stored above in the table (fig-1) is stored in a hashmap then it is stored in the form of a key-value pair. We can decide what can be the key and what can be the value. Here, it seems pretty obvious that the name of a country should be the key and its population should be the value. Why? Think about this!!!

Another example of the key-value pairs (which you might be interested in) is for a college canteen. Let's say there are two tables. One is for which day, what is the menu and other is say for the price of each item:

Day	Item
Mon	Samosa, Parantha
Tue	Pizza, pasta
Wed	Chowmin, Samosa
Thu	Chilli potato, parantha
Fri	Pizza, Veg Roll
Sat	Pasta, Chilli Potato

Item	Price (Rs)
Samosa	10
Parantha	25
Chowmin	40
Veg Roll	45
Pizza	45
Pasta	45
Chilli Potato	50

These two tables can be stored in two hashmaps. One will have both key and value as a string (menu table) and the other will have key as a string and value as an integer. So, we can have any combination of data types when we talk about hashmaps. Also, there is one important thing to note in the price of the items table above (fig). Three items, vegetarian-roll, pizza and pasta, all have a price of 45. So, different keys can have the same values. But can we have multiple pasta keys or multiple pizza keys in the hashmap? You will get to know this very soon.

## Put

Time Complexity:

$O(1)$

We can use the `put(key,value)` function to put some values into the hashmap. There are two possible cases while we are using the `put(key,value)` function in a hashmap.

1. The key is not present: If the key is not present in the hashmap it will get inserted.
2. The key is already present: If the key is already present, we cannot insert the same key again. The value of the existing key will get updated in the hashmap.

Let us insert the population of various countries into the hashmap:

```
import java.io.*;
import java.util.*;

public class Main{

    public static void main(String[] args)
    {
        HashMap<String, Integer> hm=new HashMap<>();
        hm.put("India", 1391);
        hm.put("China", 1398);
        hm.put("USA", 329);
        hm.put("Indonesia", 268);
        System.out.println(hm);

        //When we try to insert the same key with different value
        hm.put("Indonesia", 270);
        System.out.println(hm);
    }
}
```

We first try to insert the four countries with their population and print the hashmap. These will get inserted as there is no key already present with the same name. Now, we try to again insert the key "Indonesia" which is already present. So, its value will get updated. (Have a look at the output given below)

**OUTPUT**

```
{USA=329, China=1398, India=1391, Indonesia=268}
          value updated
{USA=329, China=1398, India=1391, Indonesia=270}
```

## Get a value in Hashmap

Time Complexity:

$O(1)$

We can use the `get(key)` function in a hashmap to get the value corresponding to a particular key in the hashmap. There can be two possible cases for this:

1. If the key exists: If the key exists, you will get the value of that key by using this function.
2. If the Key doesn't Exist: If the key doesn't exist, this function will return null.

Let us try to get from the hashmap that we created above, both, a value for which key exists and one for which it doesn't.

```
import java.io.*;
```

```

import java.util.*;

public class Main {

    public static void main(String[] args)
    {
        HashMap< String, Integer> hm = new HashMap<>();
        hm.put("India", 1391);
        hm.put("China", 1398);
        hm.put("USA", 329);
        hm.put("Indonesia", 268);

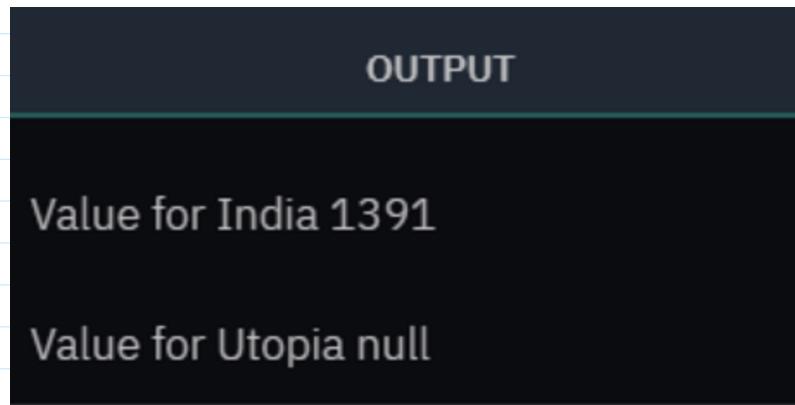
        //Printing the get(key) when key exists
        System.out.println("Value for India " + hm.get("India"));

        //Printing the get(key) when key doesn't exist
        System.out.println("Value for Utopia " + hm.get("Utopia"));

    }
}

```

When we try to get the value for key "India", it already exists and we get the value that we set earlier but key "Utopia" doesn't exist in the hashmap so we get null for it (Refer to the output given below)



**Contains Key in Hashmap**

Time Complexity:

$O(1)$

This is a Boolean function. We pass a particular key value as a parameter to this function. If the hashmap contains that key, it returns true else it returns false. So, let us implement it.

```

import java.io.*;
import java.util.*;

public class Main{

    public static void main(String[] args)
    {
        HashMap<String, Integer> hm=new HashMap<>();
        hm.put("India", 1391);
        hm.put("China", 1398);
        hm.put("USA", 329);
        hm.put("Indonesia", 268);

        System.out.println("The hash map contains China: "+hm.containsKey("China"));
        System.out.println("The hash map contains Pakistan: "+hm.containsKey("Pakistan"));
    }
}

```

We have checked whether the hashmap contains the keys "China" and "Pakistan" in it or not. We get true for the key "China" as it exists in the hashmap and we get false for the key "Pakistan" as it does not exist. (refer to the output below)

**OUTPUT**

---

The hash map contains China: true

The hash map contains Pakistan: false

### Key Set in Hashmaps

Time Complexity:

$O(1)$

We know that we have key-value pairs in a hashmap. We can get only the keys in a hashmap by using the keySet() function. The key set function returns the set of all the keys in a hashmap. We can store it in a Set container. We will study about the Set container later. For now, you can just understand that to get the Set of keys, we store it in a Set. The implementation is given below:

```

import java.io.*;
import java.util.*;

public class Main {

    public static void main(String[] args)
    {
        HashMap< String, Integer> hm = new HashMap<>();
        hm.put("India", 1391);
        hm.put("China", 1398);
        hm.put("USA", 329);
        hm.put("Indonesia", 268);

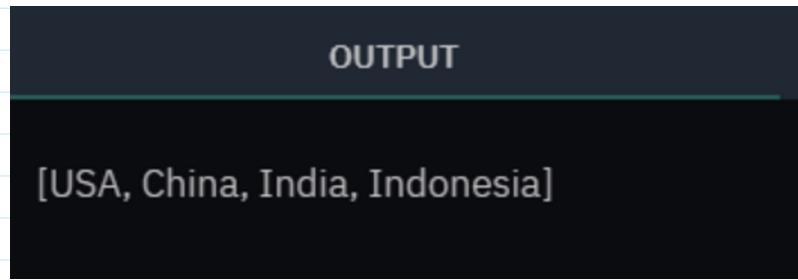
        Set< String> keys = hm.keySet();
        System.out.println(keys);
    }
}

```

```
}
```

```
}
```

We get the set of keys into the set keys container. The Set keys is displayed also. (refer to the output given below)



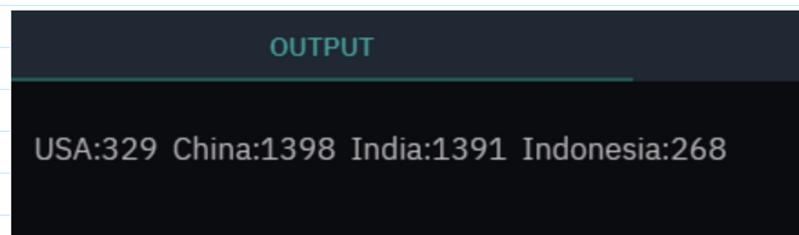
How is this keySet() function useful to us? Well, we can apply a loop on the hashmap using this. An example for the same is shown below:

Java Code (Implementation of keySet())

```
import java.io.*;
import java.util.*;
public class Main {
    public static void main(String[] args)
    {
        HashMap< String, Integer> hm = new HashMap<>();
        hm.put("India", 1391);
        hm.put("China", 1398);
        hm.put("USA", 329);
        hm.put("Indonesia", 268);

        //Using keySet() to iterate through the hashmap
        for (String key : hm.keySet())
        {
            Integer val = hm.get(key);
            System.out.print(key + ":" + val + " ");
        }
    }
}
```

We have iterated through the hashmap using the keySet() function and we print the key value pairs. The output for the same is shown below:



So, we have seen some of the basic and most commonly used functions of the hashmap. If you notice, all the functions that we have studied have  $O(1)$  time complexity and hence hashmap becomes so important and useful to us. In many questions, it will help us reduce the time complexity and improve the performance of our code.



# Highest Frequency Character

22 October 2022 12:31

```
package com.HashMap;

import java.util.HashMap;

public class HighFreqCh {
    // In this question, you are given a string str and are required to find the character with maximum frequency.
    public static void main(String[] args) {
        String str = "abcaabcd";
        HashMap<Character, Integer> hm = new HashMap<>();
        for(int i=0; i<str.length(); i++){
            if(hm.containsKey(str.charAt(i))){
                int store = hm.get(str.charAt(i));
                hm.put(str.charAt(i), store+1);
            } else hm.put(str.charAt(i), 1);
        }
        int maxF=0;
        char ch = 0;
        for(Character key: hm.keySet()){
            if(maxF<hm.get(key)){
                maxF=hm.get(key);
                ch=key.charValue();
            }
        }
        System.out.println(ch + " " +maxF);
    }
}
```

# Common Element I

22 October 2022 17:54

```
package com.HashMap;

import java.util.HashMap;

public class CommonElement {
    public static void main(String[] args) {
        String st1="abcdabcf";
        String str2="cbacbaafg";
        HashMap<Character, Integer> hm=new HashMap<>();
        for (int i = 0; i < st1.length(); i++) {
            if(hm.containsKey(st1.charAt(i))){
                int store = hm.get(st1.charAt(i));
                hm.put(st1.charAt(i),store+1);
            }else hm.put(st1.charAt(i),1);
        }
        for (int i = 0; i < str2.length(); i++) {
            if(hm.containsKey(str2.charAt(i))){
                System.out.print(str2.charAt(i));
                hm.remove(str2.charAt(i));
            }
        }
    }
}
```

Output : cbaf

## COMMON ELEMENT 2

24 October 2022 13:11

1. You are given a number  $n_1$ , representing the size of array  $a_1$ .
2. You are given  $n_1$  numbers, representing elements of array  $a_1$ .
3. You are given a number  $n_2$ , representing the size of array  $a_2$ .
4. You are given  $n_2$  numbers, representing elements of array  $a_2$ .
5. You are required to find the intersection of  $a_1$  and  $a_2$ . To get an idea check the example below:

if  $a_1 \rightarrow 1\ 1\ 2\ 2\ 2\ 3\ 5$   
and  $a_2 \rightarrow 1\ 1\ 1\ 2\ 2\ 4\ 5$   
intersection is  $\rightarrow 1\ 1\ 2\ 2\ 5$

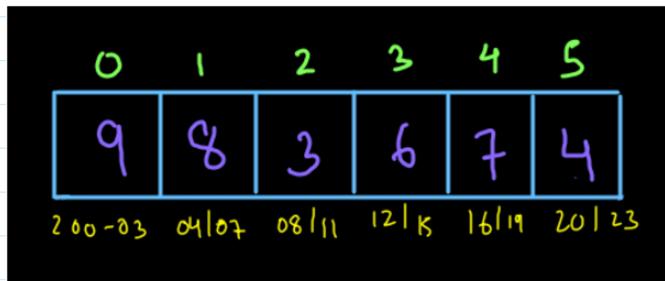
```
package com.HashMap;

import java.util.HashMap;

public class CommonElement2 {
    public static void main(String[] args) {
        String st1="qqqq";
        String str2="cqqc";
        HashMap<Character, Integer> hm=new HashMap<>();
        for (int i = 0; i < st1.length(); i++) {
            if(hm.containsKey(st1.charAt(i))){
                int store = hm.get(st1.charAt(i));
                hm.put(st1.charAt(i),store+1);
            }else hm.put(st1.charAt(i),1);
        }
        for (int i = 0; i < str2.length(); i++) {
            int store=1;
            if(hm.containsKey(str2.charAt(i))){
                store=hm.get(str2.charAt(i));
                if((store)<=0) hm.remove(str2.charAt(i));
                else{
                    hm.put(str2.charAt(i),store-1);
                    System.out.print(str2.charAt(i));
                }
            }
        }
    }
}
```

24 October 2022 13:14

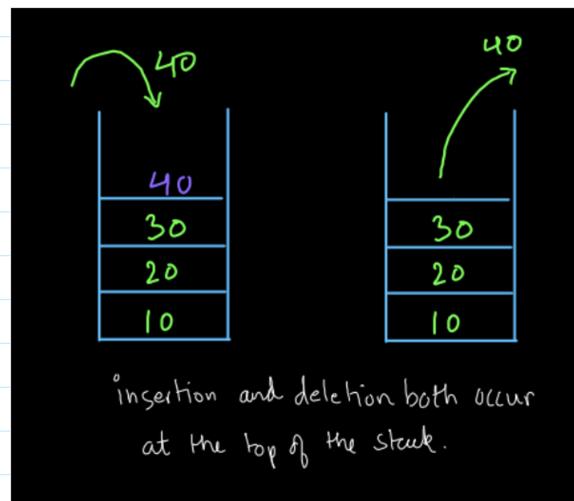
**Array:** An array is a linear data structure which stores elements of the same data type in a contiguous memory. The contiguous memory means that the memory addresses of the elements of an array are in continuation and it is not random.



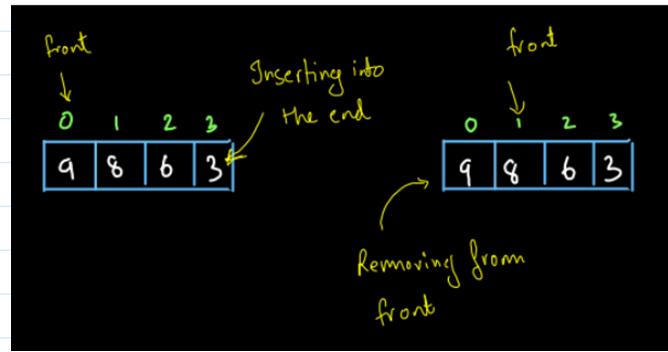
The size of int data type is 4 bytes. So, if the first element has a memory address of 200 and it takes 4 bytes i.e. 200-203, then the next element starts from the very next memory address 204.

**ArrayList:** An arraylist is just like an array. The only difference is that an arraylist is dynamic in nature i.e. it does not have a fixed size. We can enter as many elements as we want in the array list and it increases/adjusts its size automatically.

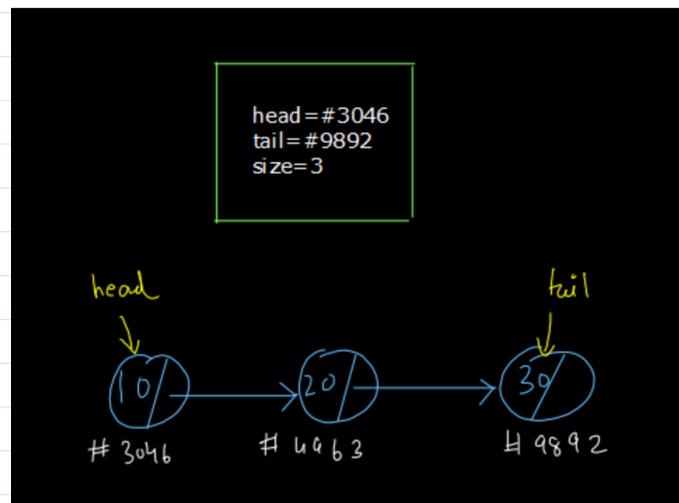
**Stack:** A stack data structure is also dynamic, just like an arraylist but it follows the LIFO principle i.e. last-in-first-out principle. The element that is inserted first into the stack comes out of it last. The elements are inserted and deleted from the top only.



**Queue:** A queue is also very similar to an arraylist because of its dynamic size property, but it follows FIFO principle i.e. first-in-first-out principle. The elements are always inserted at the rear end of the queue and always deleted from the front.



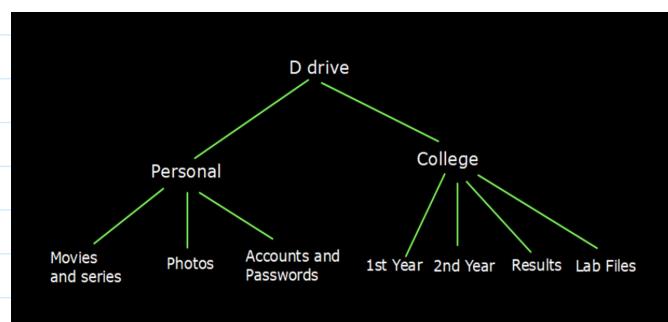
**Linked List:** The linked list data structure is also a linear data structure where the memory is not contiguous and the nodes are connected to each other via some links. These links are not physical. We store the address of the next node in the previous node and the first node is called the head and the last node is the tail.



So, what is common between all these data structures? They all are linear data structures i.e. the physical structure or the memory allocation of these data structures is linear. But, in real life, many times we do not want the data to be stored linearly. Therefore these data structures can't be enough for us.

Now the question is, "What is a tree data structure and what does it offer us?"

Let us take an example of my own laptop and the folders inside it. Have a look at the diagram given below:

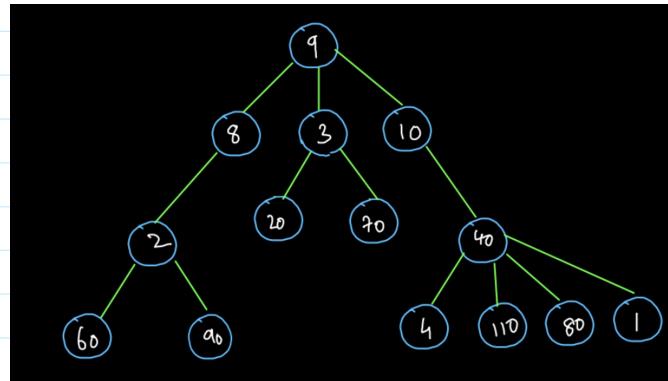


Inside the "D drive" of my laptop, I have got two folders. One of them is named personal and other is named college. The personal folder has 3 folders inside it i.e. "Movies and Series", "Photos" and "Accounts and Passwords" whereas, the college folder has 4 folders inside it as shown above. This form of data arrangement is called hierarchical arrangement of data.

The hierarchy means the generation order. As in the above example the personal folder and the college folder are inside the "D drive". So, the D drive comes first and it can be called the parent of the college folder and the personal folder. So, this type of data arrangement where hierarchy is involved can be stored using a data structure called the tree data structure.

### Generic Tree:

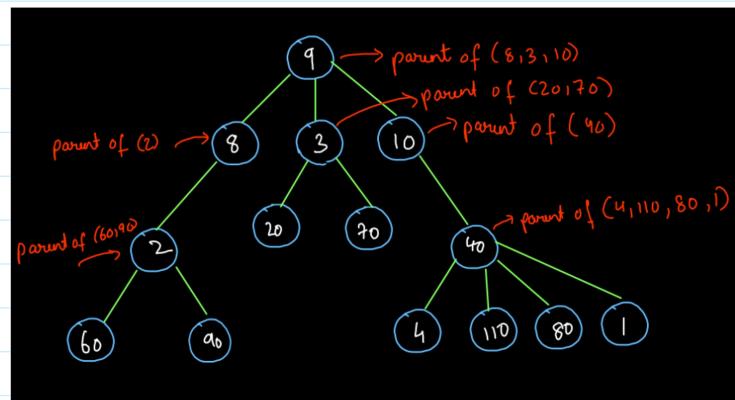
The above diagram (fig-4) shows a hierarchical structure of the folders and that structure is called a tree. As any other data structure, a tree can be an integer tree or a tree of double numbers or a tree of strings etc. A tree of integers is shown below:

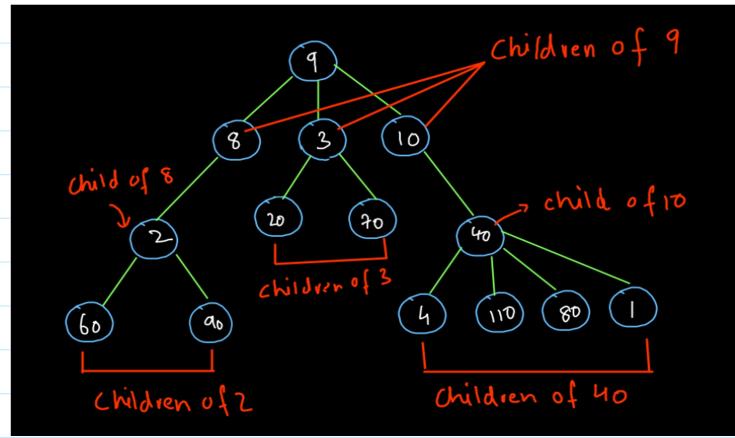


The element "9" is at the top of the hierarchy and all the other nodes are descended from it. Let us understand some keywords regarding the tree data structure and understand why this tree is called a "generic tree".

Parent Node: The node that is connected to one or many nodes in the next hierarchical level (i.e. level below this node) is called a parent node.

Child Node: The node that is connected to one node in the previous hierarchical level (i.e. level above it) is called a child node.



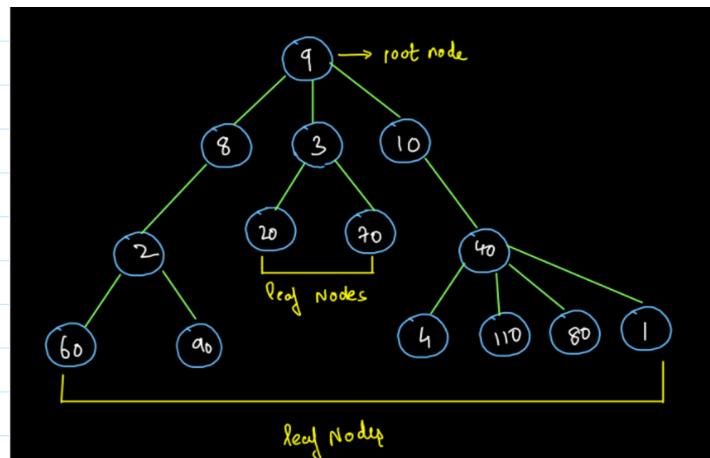


As shown in the diagrams above, 9 is the parent node of 8, 3 and 10. Similarly, 8 is the parent node for 2, 3 is the parent node for 20 and 70 and 10 is the parent node for 40 and so on.

Also, in fig-7, we can see who is a child of whom. So, we guess that you have understood till here.

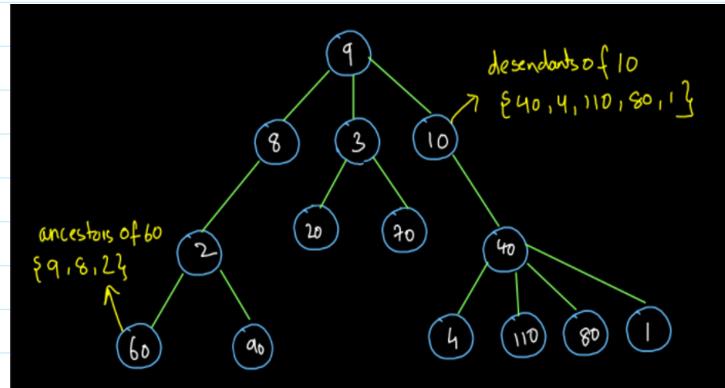
Root Node: The node at the top of the hierarchy level in a tree who has no parent node and all the other nodes descended from it is called the root node. For example: 9 is the root node in the tree shown above.

Leaf Nodes: All the nodes (not necessarily at the lowest level) that do not have any child nodes are called the leaf nodes.



Ancestors of a Node: All the nodes from which a particular node is inherited are called the ancestors of that node.

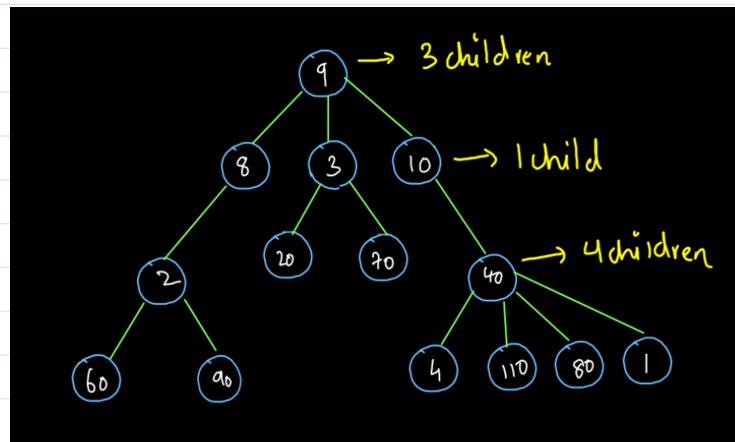
Descendants of a Node: All the nodes that descend (or inherit) from a particular node are called the descendants of that node. We can also say that all the nodes in the sub-tree descended from a node are its descendants. For example: in the diagram below, all the elements of the sub-tree that is descendant from 10 are its descendants.



Note: The root node is the ancestor of every node as every node is descended from it.  
 So, we hope that you have understood this terminology. There are a lot of more terms related to the tree data structure that we can study. We will study them as we move forward in the topic.  
 We have still not answered one question i.e. what exactly is the difference between a tree and a generic tree? So let's answer this question.

Understanding the Generic Tree:

Have a look at the diagram shown below:

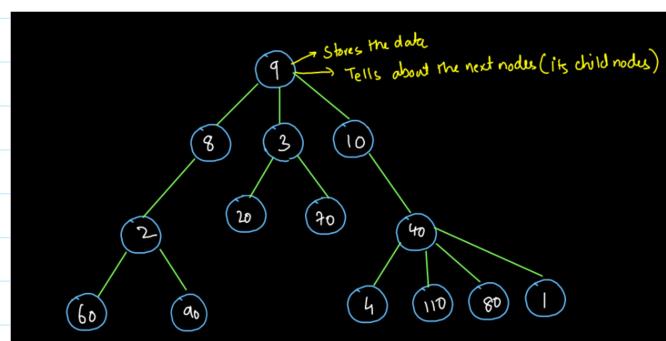


Three different nodes in this tree have 3 different numbers of child nodes. So, if the maximum number of child nodes of a tree is not fixed and it can have as many child nodes as there can be, then it is called a generic tree.

So, a generic tree is a tree where it might happen that a node would not have any child (leaf node) and a node would have 10 or 100 children. So, the maximum number of child nodes of a node is not specific rather general and hence the name is generic tree.

### Generic Tree Data Members

A tree is made up of nodes. So, before we make our own tree, we need to create our node first. What information do we get from a particular node of a generic tree?



A node in a generic tree stores its own data and also tells us about its child nodes. How will it tell us about its child nodes? It will tell us about the child node by storing the addresses of these child nodes also.

So, we do not have a single child node, rather we have multiple. So, we can use an array of nodes to store the addresses of the child nodes.

But, there is another problem. We do not know about the number of child nodes of a particular node. If a node has two children, we have to make an array of size 2 and if it has 5 children, we have to make an array of size 5. So, we have to make an array of dynamic size. Therefore, it is better to use an arraylist to achieve this functionality.

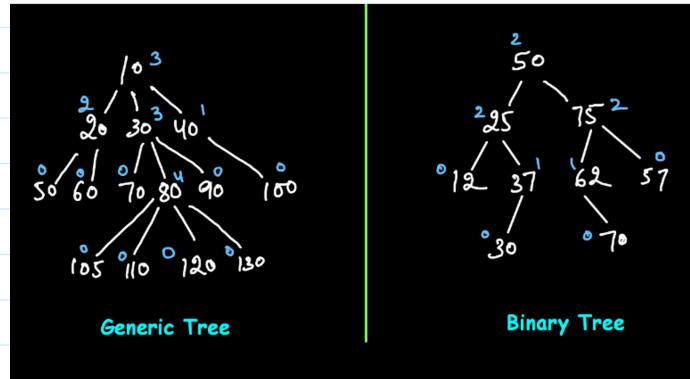
Therefore, the node will consist of its data and an arraylist of its child nodes.

Also, a tree is identified only by its root node. So, we do not need to make a separate class for generic trees, rather we can make a root variable of type node and use it directly to show a unique tree. This is shown in the code below:

# Binary Tree

26 February 2023 23:47

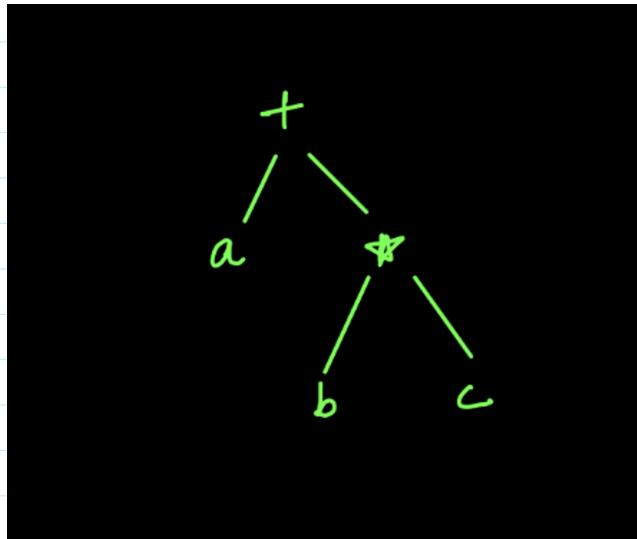
As you can see in figure 1, nodes of generic trees can have more than 2 children. But for a binary tree, each node has at most 2 children (number of children are mentioned besides each node) .



Approach :

## SOME APPLICATIONS OF BINARY TREE:

- Binary trees are used for searching and sorting as they provide us a means to store the data hierarchically.
- Also operations like insertion, deletion, and traversal can be conducted on binary trees.
- We can also solve expressions using Binary Trees.
- Say, for an expression  $a+b*c$ , its binary tree can be represented by figure 2.



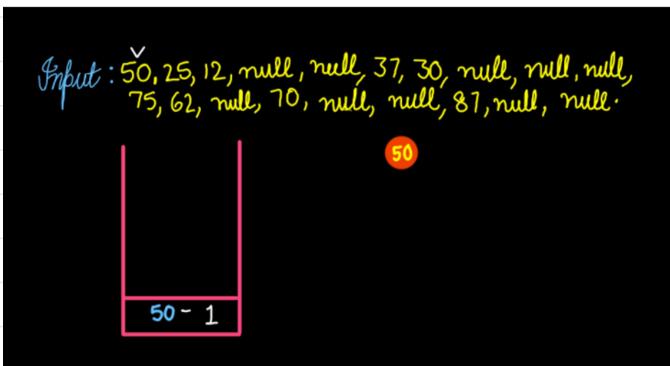
## Binary Tree Constructor

27 February 2023 00:46

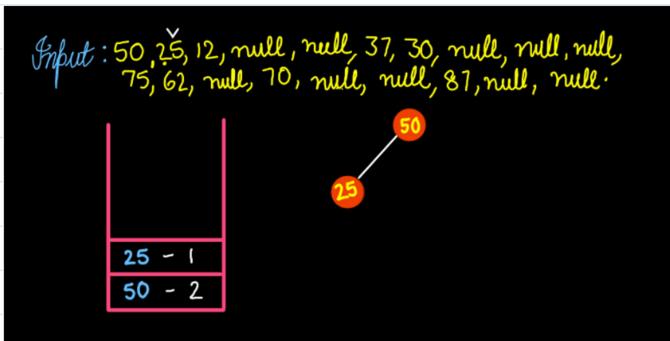
Let's take an example and understand this in a better way. Consider the below given preorder as the input.

Input (Preorder): 50, 25, 12, null, null, 37, 30, null, null, null,  
75, 62, null, 70, null, null, 87, null, null.

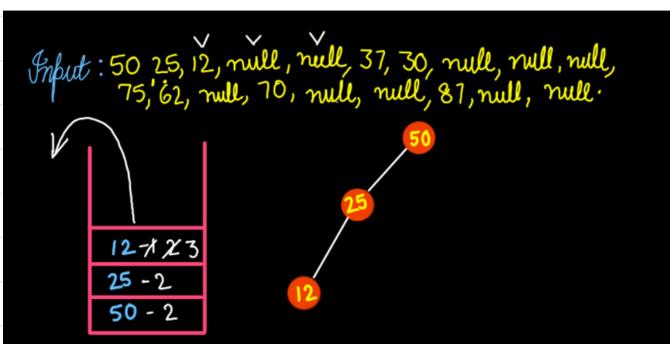
Initially the first node (50) will be pushed into the stack with corresponding value of stage to be 1.



Since the stage of the top most (50) element of the stack is 1, this implies that the next element (25) of the input array will be the left value of the node. So we push this into the stack with stage value 1. And increment the stage value of 50, making it 2.

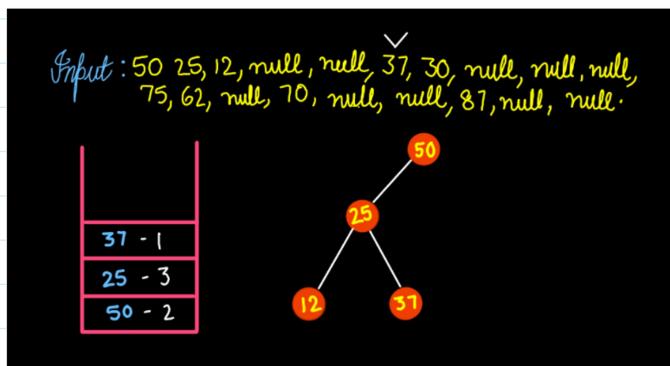


Now, the stage of top most element (25) of stack is 1, this implies that the next element (12) of the input array will be the left value of the node (25). So we push this into the stack with stage value 1. And increment the stage value of 25, making it 2.

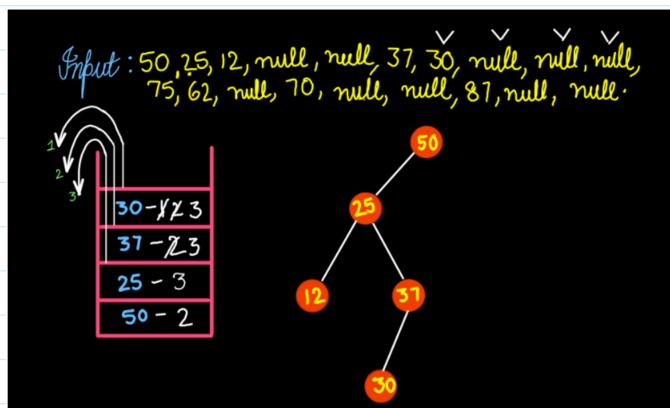


Now, the stage of top most element (12) of stack is 1, this implies that the next element null will be the left of the node (12). Since it is a null we only increment the stage value of 12, making it 2 and we move to the next element. Now, the stage of top most element (12) of stack is 2, this implies that the next element null will be the right of the node (12). Since it is a null we only increment the stage value of 12, making it 3. As the stage of topmost element (12) becomes 3, 12 pops out of the stack and we move to the next element.

Now, the stage of top most element (25) of stack is 2, this implies that the next element (37) of the input array will be the right value of the node (25). So we push this into the stack with stage value 1. And increment the stage value of 25, making it 3.



Now, the stage of top most element (37) of stack is 1, this implies that the next element (30) of the input array will be the left value of the node (37). So we push this into the stack with stage value 1. And increment the stage value of 37, making it 2.

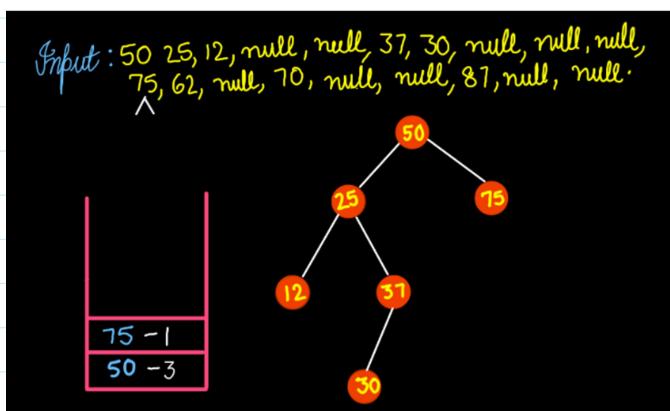


Now, the stage of top most element (30) of stack is 1, this implies that the next element null will be the left of the node (30). Since it is a null we only increment the stage value of 30, making it 2 and we move to the next element. The stage of top most element (30) of stack is 2, this implies that the next element null will be the right of the node (30). Since it is a null we only increment the stage value of 30, making it 3. As the stage of topmost element (30) becomes 3, 30 pops out of the stack.

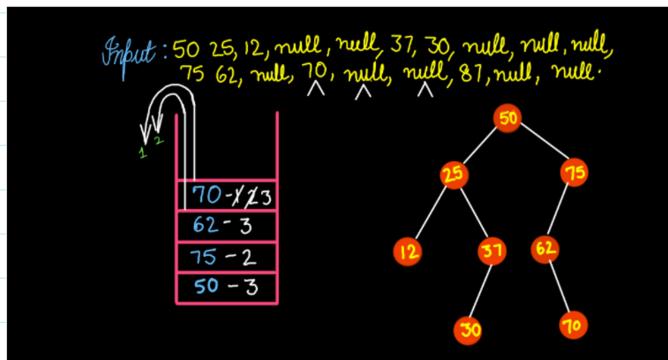
Now, the stage of the top most element (37) of stack is 2, this implies that the next element null will be the right value of the node (37).. And increment the stage value of 37, making it 3. As the stage of topmost element (37) again becomes 3, 37 pops out of the stack.

The stage of the top most element (25) of stack is 3, this implies that 25 also pops out of the stack.

Now, the stage of the top most (50) element of the stack is 2, this implies that the next element (75) of the input array will be the right value of the node. So we push this into the stack with stage value 1. And increment the stage value of 50, making it 3.



Now, the stage of top most element (62) of stack is 1, this implies that the next element null will be the left of the node (62). Since it is a null we only increment the stage value of 62, making it 2 and we move to the next element. The stage of the top most element (62) of stack is 2, this implies that the next element 70 of the array will be the right of the node (62). So we push this into the stack with stage value 1. And increment the stage value of 62, making it 3.

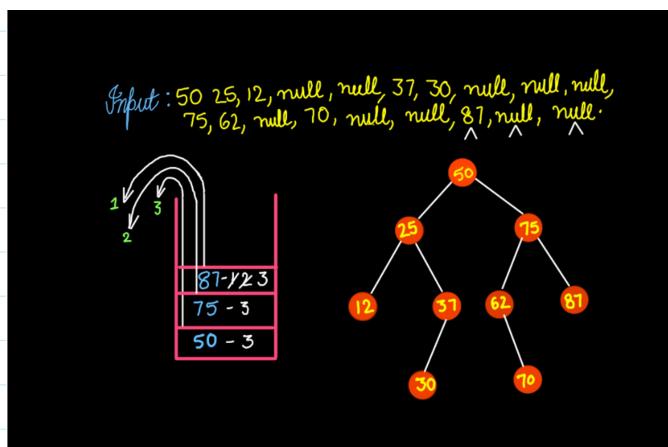


Now, the stage of top most element (70) of stack is 1, this implies that the next element null will be the left of the node (70). Since it is a null we only increment the stage value of 70, making it 2

The stage of top most element (70) of stack is 2, this implies that the next element null will be the right of the node (70). Since it is a null we only increment the stage value of 70, making it 3. As the stage of the topmost element (30) becomes 3, 70 pop out of the stack.

Now, the stage of top most element (62) of stack is 2, this implies that the next element null will be the right value of the node (62). And increment the stage value of 62, making it 3. As the stage of topmost element (62) also becomes 3, 62 pops out of the stack.

The stage of top most element (75) of stack is 2, this implies that the next element (87) of the input array will be the right value of the node (75). So we push this into the stack with stage value 1. And increment the stage value of 75, making it 3.



Now, the stage of top most element (87) of stack is 1, this implies that the next element null will be the left of the node (87). Since it is a null we only increment the stage value of 87, making it 2.

Now, the stage of top most element (87) of stack is 2, this implies that the next element null will be the right of the node (87). Since it is a null we only increment the stage value of 87, making it 3. As the stage of the topmost element (87) becomes 3, 87 pops out of the stack.

The stage of the top most element (75) of the stack is 3, this implies that it also pops out of the stack.

Also, the stage of the top most element (50) of the stack is 3, this implies that 50 also pops out of the stack.

## CODE

```
public static class Pair {
    Node node;
    int state;

    Pair(Node node, int state) {
        this.node = node;
        this.state = state;
    }
}
```

In the above code snippet, a class Pair has been created. It has a Node and int as two of its properties.

Then we have defined a constructor with Node and state as its parameters.

```
public static Node construct(Integer[][]arr) {
    Node root = new Node(arr[0]);           //1
    Stack<Pair>st = new Stack<>();        //2
    Pair root_pair = new Pair(root,1);      //3
    st.push(root_pair);
    int idx = 1;                          //4
    while(st.size() > 0) {                //5
        Pair top = st.peek();             //6
    }
```

1. A node is defined as the root which stores the value of an array at index 0. Since the array is arranged in preorder, the first value is supposed to be the root's data.
2. Then a stack, of type Pair, is created to carry out a whole and heavy mission.
3. A new pair is defined, root is stored as node and 1 as its stage.
4. Then the pair is pushed in the stack before the real operation starts.
5. An integer named idx is defined and initialized with 1, in order to access the input array.
6. Now we run a while loop, until the stack empties out.
7. To start with, we store the top of the stack in a variable top of type pair obviously so that we can access it's information further.

```
if(top.state == 1) {           //1
    //waiting for left child
    top.state++;
    if(arr[idx] != null) {     //2
        Node lc = new Node(arr[idx]); //3
        top.node.left = lc;       //4
        Pair lcp = new Pair(lc,1); //5
        st.push(lcp);
    }
    idx++;                   //6
}
```

1. Moving further we first check, if the stage of Pair top equals to 1. If it is true then further statements get executed otherwise we check the next condition.
2. As soon as control enters the if condition, we increment the top.state.
3. Then we check whether arr(idx) equals null or not.
4. If it doesn't equal null then we firstly define a new Node lc (left child) and pass arr(idx) as it's data value.
5. After that we set the Node lc as the left of Node node.
6. Furthermore, we define a pair lcp (left child pair) and set lc as it's Node and 1 as its stage.
7. Then we push this pair into the stack.
8. And finally increment idx.

```
else if(top.state == 2) {           //1
    //waiting for right child
    top.state++;
    if(arr[idx] != null) {         //2
        Node rc = new Node(arr[idx]); //3
        top.node.right = rc;       //4
        Pair rcp = new Pair(rc,1); //5
        st.push(rcp);
    }
    idx++;                   //6
}
```

1. Moving further, if the initial 'if' condition wasn't true then we now check if the stage of Pair top equals to 2. If it is true then further statements get executed otherwise we skip this and jump to the next part.
2. As soon as control enters the if condition, we increment the top.state.
3. Then we check whether arr(idx) equals null or not.
4. If it doesn't equal null then we firstly define a new Node rc (right child) and pass arr(idx) as it's data value.
5. After that we set the Node rc as the right of Node node.
6. Furthermore, we define a pair rcp (right child pair) and set rc as it's Node and 1 as its stage.
7. Then we push this pair into the stack.
8. And finally increment idx.

```

        else if(top.state == 3) {           //1
            st.pop();                     //2
        }
    return root;                      //3
}

```

1. If none of the above mentioned conditions was true then we check if stage of top equals 3.
2. If the condition is true then we pop out the top of the stack.
3. After coming out of the while loop we return the root.

```

import java.util.*;

public class Main {
    public static class Node {
        int data;
        Node left;
        Node right;
        Node() {
        }
        Node(int data) {
            this.data = data;
        }
    }
    public static class Pair {
        Node node;
        int state;

        Pair() {
        }

        Pair(Node node, int state) {
            this.node = node;
            this.state = state;
        }
    }

    public static Node construct(Integer[] arr) {
        Node root = new Node(arr[0]);

        Stack<Pair> st = new Stack< >();
        Pair root_pair = new Pair(root, 1);

        st.push(root_pair);
        int idx = 1;

        while (st.size() > 0) {
            Pair top = st.peek();

            if (top.state == 1) {
                //waiting for left child
                top.state++;
                if (arr[idx] != null) {
                    Node lc = new Node(arr[idx]);
                    top.node.left = lc;

                    Pair lcp = new Pair(lc, 1);
                    st.push(lcp);
                }
            }
        }
    }
}

```

```

        }
        idx++;
    }
    else if (top.state == 2) {
        //waiting for right child
        top.state++;
        if (arr[idx] != null) {
            Node rc = new Node(arr[idx]);
            top.node.right = rc;

            Pair rcp = new Pair(rc, 1);
            st.push(rcp);
        }
        idx++;
    }
    else if (top.state == 3) {
        st.pop();
    }
}
return root;
}

public static void display(Node node) {
    if (node == null) {
        return;
    }

    String str = " <- " + node.data + " -> ";

    String left = (node.left == null) ? ":" : "" + node.left.data;
    String right = (node.right == null) ? ":" : "" + node.right.data;

    str = left + str + right;

    System.out.println(str);

    display(node.left);
    display(node.right);
}

public static void main(String[] args) {
    Integer[] arr = {50, 25, 12, null, null, 37, 30, null, null, null, 75, 62, null, 70, null, null, 87, null, null};

    Node root = construct(arr);
}
}

```

# To do

14 June 2022 02:47

- BS last part
- BS in 2d Arrays
- String notes
- All sorting algorithm

# doubt

07 May 2022 18:52

1. How other data type except string stored in memory?
2. Prim vs non-prim -- how stored in memory

# content

13 February 2022 14:05

- 1. The given num is prime or not?
- 2. Given an array nums of integers, return how many of them contain an even number of digits.
- 3. Armstrong no
- 4. sum and reverse number
- 5. palindromic number

## 1. The given num is prime or not?

```
//code --
package com.roshan;

import java.util.Arrays;
import java.util.Scanner;

public class Demo2 {
    public static void main(String[] args) {

    /*

```

Prime number : the number is divisible by 1 and itself only

Method 1:

read the number n.

Check the divisibility of the number from 2 to n/2.

If number is divisible by any of the numbers above . It isn't prime

Else it is prime.

Method 2:

We have used method 2

1) suppose the number is 141

2) find nearest square root to it. In this case it is  $\sqrt{144}=12$ .

3) Now check the divisibility of the number by prime numbers from 2 to 12. Here check whether 141 is divisible by 2,3,5,7,11.

\*/

```
boolean flag=true;
Scanner in=new Scanner(System.in);
long num=in.nextLong();
int newNum;
```

```
int sr = (int)Math.sqrt(num);
```

```
// Calculate perfect square
```

```
int a = sr * sr;
int b = (sr + 1) * (sr + 1);
```

```
// Find the nearest
```

```
if ((num - a) < (b - num))
    newNum = a;
else
    newNum=b;
```

```

        for(int numberToCheck = 2; numberToCheck<=Math.sqrt(newNum);numberToCheck++){
            boolean isPrime= true;
            for (int factor = 2; factor < numberToCheck/2; factor++) {
                if(numberToCheck%factor==0){
                    isPrime=false;
                    break;
                }
                if(isPrime && num%numberToCheck==0){
                    flag=false;
                }
            }
            System.out.println(flag);
        }

    }
    //this long process we have used bcz now for long number we can quickly find prime or not
}

```

## 2. Given an array nums of integers, return how many of them contain an even number of digits.

```

package com.kunal;
// https://leetcode.com/problems/find-numbers-with-even-number-of-digits/
public class EvenDigits {
    public static void main(String[] args) {
        int[] nums = {12,345,2,6,7896};
        // System.out.println(findNumbers(nums));

        System.out.println(digits2(-345678));
    }
    static int findNumbers(int[] nums) {
        int count = 0;
        for(int num : nums) {
            if (even(num)) {
                count++;
            }
        }
        return count;
    }

    // function to check whether a number contains even digits or not
    static boolean even(int num) {
        int numberOfDigits = digits(num);
        /*
        if (numberOfDigits % 2 == 0) {
            return true;
        }
        return false;
    }
}

```

```

    */
    return numberOfDigits % 2 == 0;
}

static int digits2(int num) {
    if (num < 0) {
        num = num * -1;
    }
    return (int)(Math.log10(num)) + 1;
}

// count number of digits in a number
static int digits(int num) {

    if (num < 0) {
        num = num * -1;
    }

    if (num == 0) {
        return 1;
    }

    int count = 0;
    while (num > 0) {
        count++;
        num = num / 10; // num /= 10
    }

    return count;
}

```

}

### 3. Armstrong no.

```

package com.roshan;

import java.util.Scanner;

public class Armstrong3_11 {
    public static void main(String[] args) {

        /*
        An armstrong number is a number that is the sum of its own digits each raised to the power of the
        number of digits.
        For example
        9 = 91 = 9
        371 = 33 + 73 + 13 = 27 + 343 + 1 = 371
        8208 = 84 + 24 + 04 + 84 = 4096 + 16 + 0 + 4096 = 8028
    }
}

```

If the input is given as command line arguments to the `main()` as [153] then the program should print the output as:

Cmd Args : 153

The given number 153 is an armstrong number

If the input is given as command line arguments to the `main()` as [25] then the program should print the output as:

Cmd Args : 25

The given number 25 is not an armstrong number

\*/

```
Scanner in=new Scanner(System.in);
int num=in.nextInt();
int arm=0;
int digits;
int temp=num;
if(num<0){
    digits=num*-1;
}else{
    digits=(int)(Math.log10(num))+1; //no of digits in given num
}
while(temp>0){
    arm+=Math.pow(temp%10,digits);
    temp/=10;
}
if(num==arm){
    System.out.println("the given num "+num+" is armstrong num");
}else{
    System.out.println("the given num "+num+" is not a armstrong number");
}
}
```

#### 4. sum and reverse number

```
package com.que;

import java.util.Scanner;

public class SumAndReverseNumber {
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int num=in.nextInt();
        sumAndReverseANumber(num);
    }
    public static void sumAndReverseANumber (int number){
        int temp=number;
        int rem,rev=0,sum=0;
        while(temp>0){
            rem=temp%10;
            rev=rev*10+rem;
            temp=temp/10;
        }
        System.out.println("sum = "+sum);
        System.out.println("reverse = "+rev);
    }
}
```

//rev

```

    //rev
    rem=temp%10;
    rev=rev*10+rem;
    temp/=10;

    //sum
    sum+=rem;
}

System.out.println("Sum of digits : "+sum);
System.out.println("Reverse : "+rev);
}
}

```

## 5. [Count Odd Numbers in an Interval Range \(LC1523\)](#)

```

package com.que;

public class CountOddNum {
    public static void main(String[] args) {
        int low=1;
        int high=7;
        System.out.println("The no of odds in the given interval "+ countOdds(low,high));
        System.out.println("The no of evens in the given interval "+ countEvens(low,high));
    }

    static int countOdds(int low, int high){
        //to find the no of odds in given interval

        if(low%2==0 && high%2==0)
        {
            return((high-low)/2);
        }

        return(((high-low)/2)+1);
    }

    static int countEvens(int low, int high){
        //to find the no of evens in given interval

        if(low%2!=0 && high%2!=0)
        {
            return((high-low)/2);
        }
        return (high-low)/2+1;
    }
}

```

## 6. [Largest Perimeter triangle \(LC 976\)](#)

```

package com.que;

public class LargestPerimeterTri {
    public static void main(String[] args) {

    /*
    - sum of any two sides should be greater than third side --> then only triangle possible ---> there you
    the use formula
    - where it is not possible return 0

```

Brief

nums = [2,1,2] --> 2+1>2 && 2+2>1 && 1+2>2 (T, T, T) --> 2+1+2=5  
 nums = [1,2,1] --> 1+2>1 && 1+1>2 && 2+1>1 (F, F, T) --> 0  
 \*/

```

        int[] nums={3,6,2,3};
        System.out.println(largestPerimeter(nums));
    }
    static int largestPerimeter(int[] nums){
        int temp;
        //    int a,b,c
        for (int i = 0; i < nums.length; i++) {
            for (int j = i+1; j < nums.length; j++) {
                if(nums[j]>nums[i]){
                    temp=nums[i];
                    nums[i]=nums[j];
                    nums[j]=temp;
                }
            }
        }
        for (int i = 0; i < nums.length-2; i++) {
            if(nums[i]+nums[i+1]>nums[i+2] && nums[i+1]+nums[i+2]>nums[i] && nums[i+2]+
                nums[i]>nums[i+1]) return nums[i]+nums[i+1]+nums[i+2];
        }
        return 0;
    }
}

```

7. The given number is odd or not \*\*\*

```

package com.Math;

import java.util.Scanner;

public class Even_Odd {
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int num=in.nextInt();
        System.out.println(isOdd(num));
    }
}

```

100101  
 000001  
 &

```

        System.out.println(isOdd(num));
    }

static boolean isOdd(int num) {
    return (num&1)==1;
}

```

100101  
000001 &

000001

Taking any decimal number

0	1	1	1	0
$0 \times 2^4$	$1 \times 2^3$	$1 \times 2^2$	$1 \times 2^1$	$0 \times 2^0$

Addition of even no  
is always even only

//it means whole number is depend  
upon this last binary no. if it is 0  
then even or else it is odd  
//so we will check this last no.

0 1 1 1 0	&	0 0 0 0 1
<hr/>		
0 0 0 0 0		

No & 1 -----> checking whether we are getting 0 or 1

0 0 0 0 0 //this means given no. is even

8. Find the duplicate no. in the given array

9. In array every digit have duplicate value in the array except one . Find that one?

1. To find no of digits from any value : `(int)(Math.log10(n)+1);`
2. Mid = `start + (end-start)/2;`
3. Copyrange :  
`Arrays.copyOfRange(arr,0,mid)` //it will copy from 0 to mid of arr variable
4. `ArrayList<Integer> list = new ArrayList<>();`
5. Passing ArrayList in function  
`static ArrayList<Integer> search2(int[] arr, int value, int index, ArrayList<Integer> list1)`
6. To get ascii no of any character  
`//you can do simply by adding 0`  
`char ch = 'a';`  
`sout(a+0) --> 87 //bcz by adding 0 it converted into int data type`  
 Or else you can do in this way also ->  
`char ch = 'a';`  
`sout((int)(a)); --> 87`
7. `List<List<Integer>> list = new ArrayList<>();` //subset iterative que can go and see
8. Any character subtracting by 0 it will give that num which was stored using char data type.  
 If we do typecasting we will get ascii value.

```
char no='2';
int num = no - '0';
System.out.println(no - '0'); //2
System.out.println(num); //2
```

9. `int[] arr = new int[2]; -----> [0,0]`
10. **Integer.MIN\_VALUE** ---> The Integer. MIN\_VALUE is a constant in the Integer class that represents the minimum or least integer value that can be represented in 32 bits

`Integer.MAX_VALUE` ---> The Integer. MAX\_VALUE is a constant in the Integer class that represents the maximum or most integer value that can be represented in 32 bits

```
package com;

import java.util.Arrays;

public class Temp2 {
    public static void main(String[] args) {

//        public static void arraycopy(Object source_arr, int sourcePos, Object dest_arr, int destPos, int len)
//        Parameters :
//        source_arr : array to be copied from
```

```
// sourcePos : starting position in source array from where to copy
// dest_arr : array to be copied in
// destPos : starting position in destination array, where to copy in
// len : total no. of components to be copied.

int[] array ={1,23,44,55,5,6,7,8};
int[] ans = new int[array.length-2];
System.arraycopy(array,0,ans,0,array.length-2);
System.out.println(Arrays.toString(ans));
}
```

04 February 2023 04:08

- Instance variable default value is 0

1. Screenshot saver
2. Vpn
3. For presentation
4. Blackbox like...!
5. Saying of words.

- In 2d array int[][] arr=new[size][];

↓  
row

2d Array :

- ```
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[row].length ; col++) {
        System.out.print(arr[row][col]+ " ");
    }
    System.out.println();
}
```

Instance variables are declared inside a class and are used for storing values of objects.

~ Default value is 0.

An object is an instance of a class created using the new keyword

Local variables are variables declared inside a method. Even parameters are local variables

- public class Tester {

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

```
    Book myBook = new Book();
```

```
    Book yourBook = new Book();
```

```
    Book newBook;
```

```
}
```

Instance 4  
Object = 3

```
}
```

```
class Book {
```

```
    public String name;
```

```
    public double price;
```

```
    public String author;
```

```
    public int stock;
```

```
}
```

- public class Student {

```
    public int rollNo;
```

```
    public String studentName;
```

```
    public void findGrade(int marks) {
```

```
        char grade;
```

```
if (marks > 90)
    grade = 'A';
else if (marks < 90 && marks > 80)
    grade = 'B';
else if (marks < 80 && marks > 70)
    grade = 'C';
else
    grade = 'D';
System.out.println(grade);
}
```

# Constructor

20 October 2022 00:18

A constructor with no arguments is known as a parameterless constructor. If you don't define a constructor in a class, then Java creates a default parameterless constructor and initializes the default values to the class variables based on the data type.

```
class Customer {  
    public String customerId;  
    public String customerName;  
    public long contactNumber;  
    public String address;  
}  
public class Tester {  
    public static void main(String args[]) {  
        Customer customer1 = new Customer();  
        System.out.println(customer1.customerId);  
        System.out.println(customer1.customerName);  
        System.out.println(customer1.contactNumber);  
        System.out.println(customer1.address);  
    }  
}
```

Here, the default parameterless constructor is called every time a new object is created and the default values to the member variables are assigned.

## ➤ Parameterless constructor

You can also create parameterless constructor in a class. In this case, Java does not create a separate default constructor.

```
class Customer {  
    public String customerId;  
    public String customerName;  
    public long contactNumber;  
    public String address;  
    public Customer() {  
        System.out.println("Constructor called");  
    }  
}  
public class Tester {  
    public static void main(String args[]) {  
        Customer customer1 = new Customer();  
        System.out.println(customer1.customerId);  
        System.out.println(customer1.customerName);  
        System.out.println(customer1.contactNumber);  
        System.out.println(customer1.address);  
        Customer customer2 = new Customer();  
    }  
}
```

|                    |
|--------------------|
| Constructor called |
| Null               |
| Null               |
| 0                  |
| Null               |
| Constructor called |

## ➤ Parameter constructor

Like any other method, a constructor can also accept parameters. Generally, these are the values that need to be assigned to the instance variables of the class for that object.

```

class Customer {
    public String customerId;
    public String customerName;
    public long contactNumber;
    public String address;
    Customer(String cId, String cName, long contact, String add) {
        customerId = cId;
        customerName = cName;
        contactNumber = contact;
        address = add;
    }
}

```

The parameter values need to be passed while creating the object as shown below.

```

public class Tester {
    public static void main(String args[]) {
        Customer customer1 = new Customer("C103", "Jacob", 5648394590L,
            "13th Street, New York");
        System.out.println(customer1.customerId);
        System.out.println(customer1.customerName);
        System.out.println(customer1.contactNumber);
        System.out.println(customer1.address);
    }
}

```

#### ➤ Multiple constructor in a class

A class can have multiple constructors to initialize different members. Based on the arguments passed, the respective constructor is called.

```

class Customer {
    public String customerId;
    public String customerName;
    public long contactNumber;
    public String address;
    public Customer() {
        System.out.println("Parameterless constructor called");
    }
    public Customer(String cId, String cName, long contact, String add) {
        System.out.println("Parameterized constructor called");
        customerId = cId;
        customerName = cName;
        contactNumber = contact;
        address = add;
    }
}

```

```

public class Tester {
    public static void main(String args[]) {
        Customer customer1 = new Customer("C103", "Jacob", 5648394590L,
            "13th Street, New York");
        Customer customer2 = new Customer();
    }
}

```

}

O/P:

Parameterized constructor called

Parameterless constructor called

# This keyword

04 February 2023 04:04

Consider the below constructor for Customer class.

```
public Customer(String customerId, String customerName, long contactNumber,
    String address) {
    customerId = customerId;
    customerName = customerName;
    contactNumber = contactNumber;
    address = address;
}
```

Here, the name of the instance variables of the class and the parameters passed in the constructor are the same. In such a case, the local variables (arguments of the constructor) have more priority and therefore, only the local variables will be referred inside the above constructor. To overcome this problem, we have this keyword which can be used to refer to the class members.

In Java, this is a reference variable that refers to the current object.

Observe the code given below.

```
public class Customer {
    public String customerId;
    public String customerName;
    public long contactNumber;
    public String address;
    public Customer(String customerId, String customerName, long contactNumber,
        String address) {
        this.customerId = customerId;
        this.customerName = customerName;
        this.contactNumber = contactNumber;
        this.address = address;
    }
}
```

The compiler will now be able to differentiate between the instance variables and the local variables in the above code.

this can also be used to invoke method or constructor of the current object.

Observe the code given below.

```
public class Customer {
    public String customerId;
    public String customerName;
    public long contactNumber;
    public String address;
    public Customer() {
        System.out.println("Parameterless constructor called");
    }
    public Customer(String customerId, String customerName, long contactNumber,
        String address) {
        // this() is used to invoke the constructor of the current class
        // Since no parameters are specified, parameterless constructor will be invoked
        this();
    }
}
```

```
this.customerId = customerId;
this.customerName = customerName;
this.contactNumber = contactNumber;
this.address = address;
}
public void displayCustomerName() {
    System.out.println("Customer Name : " + customerName);
}
public void displayCustomerDetails() {
    System.out.println("Displaying customer details \n*****");
    System.out.println("Customer Id : " + customerId);
    this.displayCustomerName();
    System.out.println("Contact Number : " + contactNumber);
    System.out.println("Address : " + address);
}
}
```