Language and Translator

~~Langage~~

* Language:- Acts as a "Medium of communication".

* Programming Language:- It is a set of step-by-step instructions, commands & other syntaxes to create a software. Programming Language acts as a medium of communication between humans and computers.

Programming Language
- Low Level
  - machine Lang. (Binary Lang.)
  - Assembly Language
- Middle Level
- High Level

→ # Programming Lang = Class + object + variable + Methods #

* (1) Machine Language:- in the machine Language instructions are given in the form of 0's and 1's.

→ Advantages:- (1) Processing speed is fast
(2) Execution time is less
(3) No need of translator
(4) cost is less
(5) Ex→ Binary code etc.

→ Disadvantages:- (1) Programers are not able to memorize code.
(2) Updation is difficult
(3) Debugging is also difficult
(4) Time consuming.

*(2) Assembly Language :- in assembly language instructions are given in the form of Alpha numeric symbols (mnemonic code).

→ like ADD for Addition, SUB for subtraction etc.

→ Advantages :- (1) mainly used in processors.

(2) Here we use assembler as a translator

(3) Easy to understand compare then Binary code.

→ Disadvantages :- (1) Time consuming

(2) It is not a portable language.

(3) machine dependent

(4) one line of Assembly Language is converted in multiple line of machine code.

(5) System Architecture specific Lang.

*. Portable Language means the language capable of developing software for more than one system.

* Machine Dependent Language means the language capable of developing softwares for a specific type system.

*. Middle Level Language :- comprise of some features of low level language and some features of high level language.

Ex→ C programming language etc.

→ this are hardware friendly languges.

#. High Level Language :-

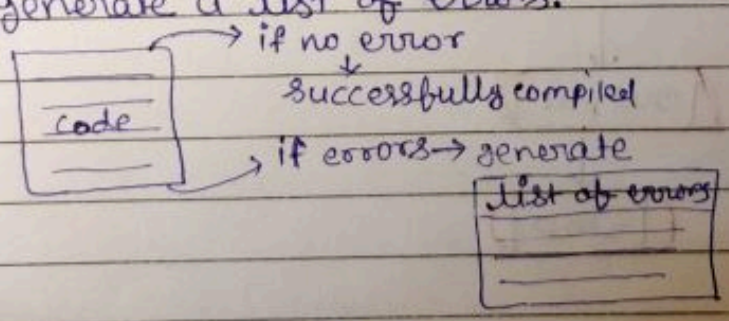
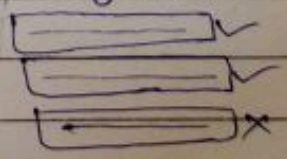
in high Level Language Instructions are given like English Language.

→ Advantages:- (1) It is a portable Language.
(2) Machine dependent
(3) Easy to memorize compared to other lang.
(4) Updation and Debugging is easier compared to other Languages.
(5) Here we use compiler and interpretor as a translator.

*. Translator :-

acts as a "Language Convertor". like
→ Assemblor :- Assembly to Binary code.
→ Interpretor :-
→ compiler :-
etc.

* Difference Between compiler and Interpreter :-

Compiler	Interpreter
① Compiler compiles whole code at a time and generate a list of errors.	Interpreter interpets the code line by line.
	
② Processing speed is fast	(2) Processing speed is slow
③ Execution time is less	③ Execution time is more.

ALGORITHMS & FLOWCHART

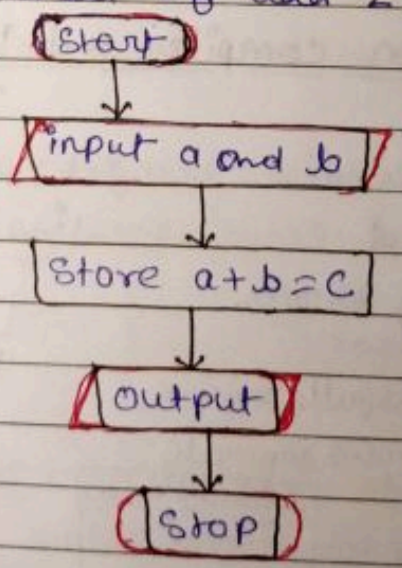
* **Algorithm :-** It is a sequential, finite, and non-complex set of steps to solve a perticular problem.

→ Example :- (1) Algo to add 2 numbers →

Step 1 - start
Step 2 - read a and b
Step 3 - C = a+b (store a + b)
Step 4 - Output C
Step 5 - Stop

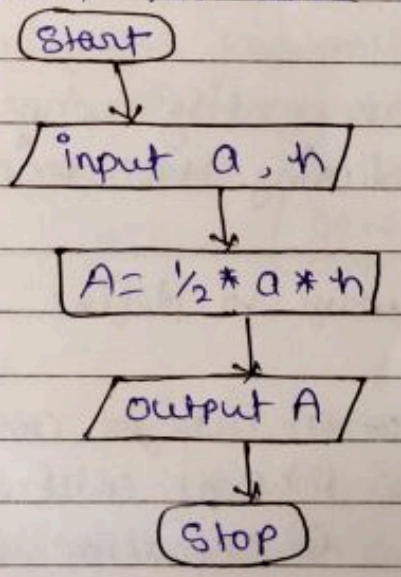
* **Flowchart :-** Pictorial Representation or Diagramatic representation or graphical representation of an algorithm to clearly the flow of control of program is known as flowchart.

→ Example :- (1) flowchart of add 2 numbers →



Symbol	Name	Example
	Terminal	Start, Stop
	Input/Output	I/O, Output
	Process	C = a + b
	Condition/decission/(diamond)	if a > b
	flowlines	
	Connector	

→ Example:- ② Flowchart for area of triangle:-



TOKENS

* Tokens :- "Smallest unit of Information."
token includes - (1) Constants
(2) keywords
(3) Variables
(4) Identifiers
(5) Seprators
(6) Operators.

(1)
* Constant :- It is an entity whose value can not
(Literal) change during the exicution of program
value is fixed.
→ There are different ways to define constant :- Java - final keyword
C - const

→ Types of constants :- (1) Integer constant
(2) Real constant
(3) Charactor constant

(1) Integer constant :- (1) contains atleast one digit
(2) may be +ve or -ve
(3) by default it considered as +ve
(4) Decimal Part is absent

(2) Real Constant :- (1) contains atleast one digit
(2) may be +ve or -ve
(3) by default It considered as +ve
(4) Decimal part is present.

(3) Character Constant :- (1) Spcial Symbol, '*' '$' '#' ';' etc.
(2) digits (0 to 9)
(c) Alphabets (a to z and A to Z)

(2)
*. Keywords :- keywords are reserved words, whose meaning is already defined. we can't use keywords as a variable name.

Ex. →
→ in C programming → 32 keywords
→ in C++ → 63 keywords
→ in Java → 51 keywords { 49 are in use
goto, const } not in use

(3)
*. Identifiers :- user defined names.
variable → Lower case
constant → Upper case
functions → Camelcase notation.

(4)
*. Separators :- , ; { } ( ) [ ]

(5)
*. Variables :- variable is the name given to the memory location where value may change during exicution of program.
Variables acts as a container.
Variables are mutable.

→ Types of variables in C :-

(1) Local variable - whose validation is within function.

(2) Golabal variable - whose valid throughout the program.

(3) Pointer variable - whose store the address of memory.

→ Types of variables in C++ :-

(1) Local variable

(2) Global variable

(3) Pointer variable

(4) Static variable

(5) Instance variable (multiple copies)

→ Types of variables in Java :-

(1) Local variable

(2) Instance variable

(3) Static variable

(4) Reference variable
- Static Ref variable.
- Local Ref. variable
- instance Ref. variable.

* Rules to define variables :-

(1) variables name can't started with digits

(2) we can use underscore as a special symbol in defining variable name

(3) where as we can use underscore and $ (dollar) as a variable name in Java.

(4) we need coma seprator to separate two or more variables

(5) By conversion we use lower case to define variables and uppercase to define constants.

* Operators :- Operators are symbols which are used to perform some operations on one or more than one operand (variable or constant).

→ Types:- (1) Assignment Operator
(2) Arithematic Operator
(3) Relational Operator
(4) Compound or shorthand or Arth. Ass. operator
(5) Logical Operator
(6) Increment / Decrement Operator
(8) Bitwise Operator
(8) Conditional operator

(1) Assignment Operators :-

it simply assigns right hand side value to left hand side variable.

Ex→

```
                  assign
int a = 100 → value
datatype  variable  assignment operator
```

(2) Arithmetic Operators :-

Performs Arithmetic Operations.

Symbol	Name	Functanality (if A = 5 & B = 2)
+	Addition	A+B = 5+2 = 7
−	Subtraction	A-B = 5-2 = 3
*	Multiplication	A*B = 5×2 = 10
/	Division	A/B = 5/2 = 2 (quotient)
%	modulus	A%B = 5%2 = 1 (Reminder).

int → withoughtt Decimal
float → Decimal

input	input	output
int	int	int
int	float	float
float	int	float
float	float	float

(3) Relational Operator :-

Symbol	Name	Functanality (if A = 5, B = 2)
<	less than	A<B ⇒ 5<2 ⇒ 0 (false)
>	greater than	A>B ⇒ 5>2 ⇒ 1 (true)
<=	less than equals to	A<=B ⇒ 5<=2 ⇒ 0 (false)
>=	greater than equals to	A>=B ⇒ 5>=2 ⇒ 1 (true)
==	comparison operator	A==B ⇒ 5==2 ⇒ 0 (false)
!=	Not equals to	A!=B ⇒ 5 ≠ 2 ⇒ 1 (true)

(4) Compound or Shorthand or Arithematic Assignment operator :-

If you want to perform Arithematic operation on a variable and set the result inside that variable itself then you can use compound operators.

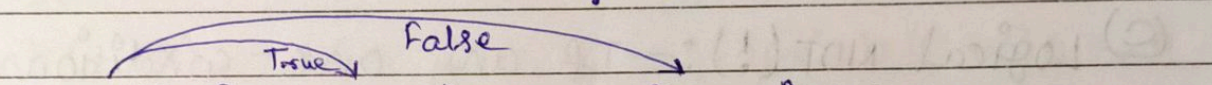
+= , -= , *= , /= , %=

(Arithematic + Assignment) = ~~Compound~~ Compound.

Symbol	Functanality (if a = 10
+=	a += 10 ⇒ a + 10 = a ⇒ a = 20
-=	a -= 10 ⇒ a - 10 = a ⇒ a = 0
*=	a *= 10 ⇒ a * 10 = a ⇒ a = 10
/=	a /= 10 ⇒ a/10 = a ⇒ a = 1
%=	a %= 10 ⇒ a%10 = a ⇒ a = 0.

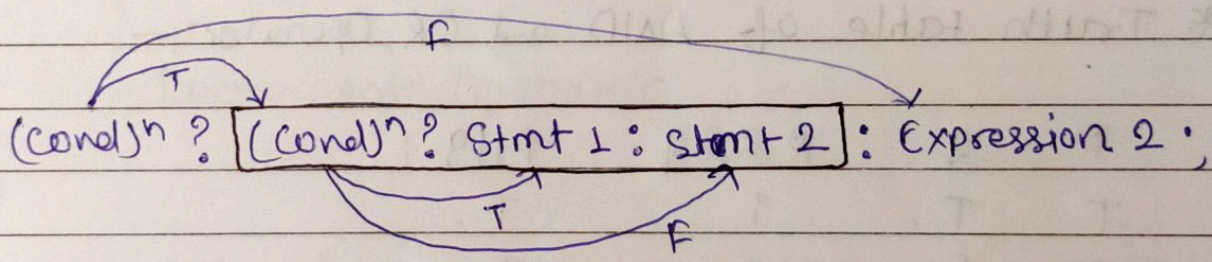
5) Conditional Operators :-

? :

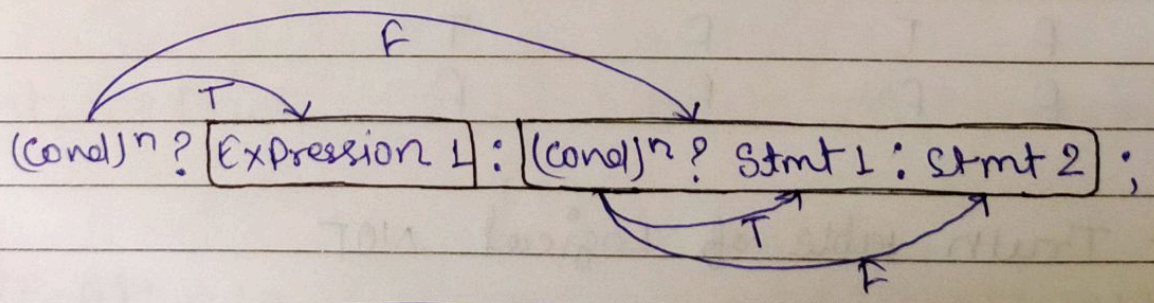


→ Nested cases :-

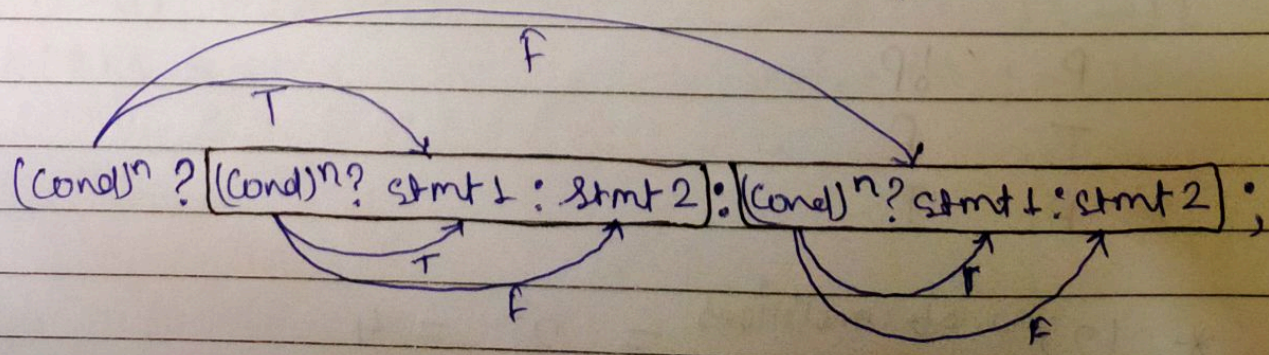
(1)



(2)



(3)



(6) Logical Operators:-

(a) Logical AND (&&):- when at the conditions are true, then the only resultant will be true.

(b) Logical OR (||):- If any one condition is true then the resultant will be true.

(c) Logical NOT (!):- If any one condition is false then the resultant will be false.

* Truth table of AND and OR Operator -

p	q	p && q	p || q
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

* Truth table of Logical NOT

p	!p
T	F
F	T

* $(2)^{\text{no. of conditions}} = 2^2 = 4$

$= 2^3 = 8$

- - - - -

- - - - -

(7) Increment / Decrement Operator :-

(++)
Increment Operator

Preincrement (++a)

Increment → Print

if a = 10
print (++a); → 11
print (a); → 11

Postincrement (a++)

Print → Increment

if a = 10
print (a++); → 10
print (a); → 11

~~(8) Bitwise~~

(--)
Decrement Operator

Predecrement (--a)

Decrement → Print

if a = 11
print (--a); → 10
print (a); → 10

Post Decrement (a--)

Print → Decrement

if a = 11
print (a--); → 11
print (a); → 10

(8) Bitwise Operator :-

0 → false
1 → true

(a) Bitwise AND (&)
(b) Bitwise OR (|)
(c) Bitwise XOR (^)
(d) Bitwise leftshift (<<)
(e) Bitwise Rightshift (>>)
(f) ~~Compute~~ Bitwise complement (~)

* Decimal to Binary →

(1) First Technique

$(9)_{10} = (1001)_2$

2	9	1
2	4	0
2	2	0
	1	

backtrack

(2) Second Technique

	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	1024	512	256	128	64	32	16	8	4	2	1
9 =								1	0	0	1
10 =								1	0	1	0
67 =					1	0	0	0	0	1	1
189 =			1	0	1	1	1	1	0	1	

(a) Bitwise AND (&) :- if a = 9
b = 10

print (a&b); → 8

1001 = 9
& 1010 = 10
1000 = 8

(b) Bitwise OR (|) :- if a = 9
b = 10

print(a|b); → 11

1001 = 9
| 1010 = 10
1011 = 11

(c) Bitwise XOR (^) :- same bit = 0
opposit bit = 1

if a = 9 , b = 10
print(a ^ b); → 3

1001 = 9
^ 1010 = 10
0011 = 3

(d) Bitwise Leftshift (<<) :-

if num = 4 , binary = 100
8 bit repⁿ : 00000100

num << 2
print(num); → 16 } →

num << 2
00000100
00100000 = 16

(e) Bitwise Rightshift (>>) :-

if num = 4 , binary = 100

num >> 2;
print(num); → 1 } →

num >> 2
00000100
00000001 = 1

(f) Bitwise Complement (~) :- ~n = -(n+1)

if n = 10 ; ~n = -(10+1) = -11

if n = 10
print(~n); → -11

(9) Specific Operator (for C):-

(1) , comma Operator
(2) sizeof () operator
(3) Pointer → & (address of) (ampresent)
→ * (value of) (astrick)

etc.

(9) Specific Operator (for C++):-

(1) << insertion Operator
(2) >> Extraction Operator
(3) :: Scope Resolution Operator
(4) new memory allocation operator
(5) delete memory release operator
(6) setw field width operator
(7) endl line feed operator
etc.

(9) Specific Operator (for Java):-

(1) misc operator
(2) . dot operator
(3) new operator
(3) instanceof operator
(4) ?: conditional operator.

#. PROCEDURE ORIENTED PROGRAMMING LANGUAGE

* POP :-

It is a way of writting a list of instructions for the computer to follow, and this instructions are assigned in a group and this groups are known as functions.

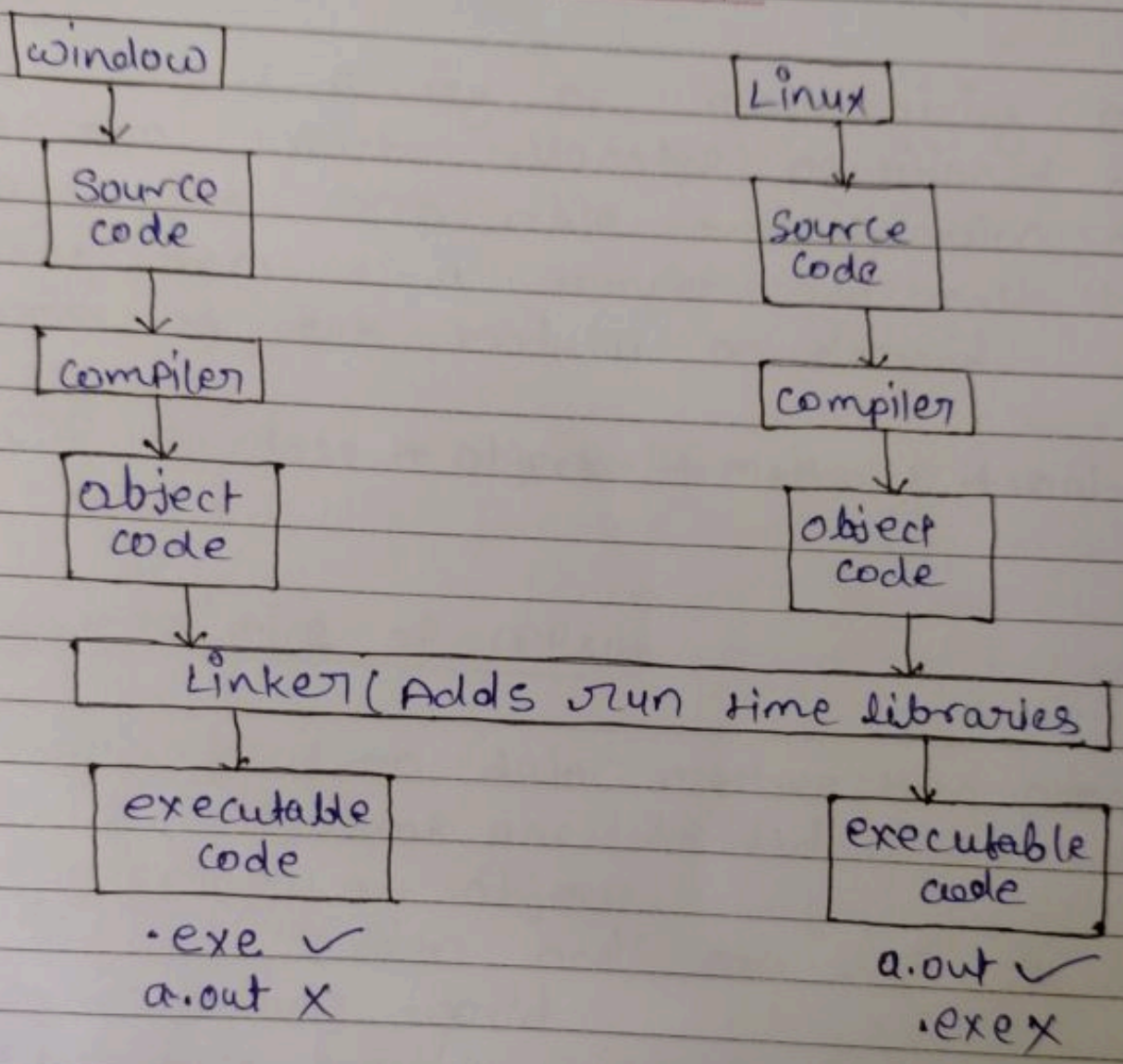
*POP = Methods + Variables.

*. Characteristics of POP Language :-

(1) mainly emphasise is on doing things.
(2) Large programs are devided into part are known as functions.
(3) data revolves arround the system.
(4) functions shares the global data.
(5) functions can also transform data.
(6) follows top down approach in program design.

Ex→ C is a POP Language.

* C / C++ Platform dependent:-



OBJECT ORIENTED PROGRAMMING LANGUAGE

#. OOP :-

It is a way of modularizing a program which allocates partitioned memory area for both data and function and using them as a templet and call the copies of such modules on demand.

OOP = Class + objects + Methods + variables.

*. Characteristics of OOPs:-

(1) mainly focus on "data" rather than proccedure.
(2) Large Programs are devided into what are known as objects.
(3) Data is hidden and can not be accessible from outside world.
(4) New data can be added any where whenever required.
(5) Objects may communicates with each other with the help of functions.
(6) follows Bottom-up approach.

* Simula is the first OOP s language.
* Smalltalk is the first purely objet oriented programming languge.

JAVA PROGRAMMING LANGUAGE

* JAVA :-

Java is a general purpose, imperative, case sensitive, class based, ~~etc~~ strictally typed, object oriented and highly syntax based programming Language.

Along with this it is also a computing platform. Here computing platform means allocation of programs which are allows Developers to develope and run Java applications.

Computing platform is the collection of libraries, compiler and execution engine.

* Types of Applications which we can create with the help of JAVA :-

Application of Java:-

(1) Standalone Application or windows based app. or Desktop Applications:-

* Technology used:-

AWT (Abstract windows toolkit).
SWING

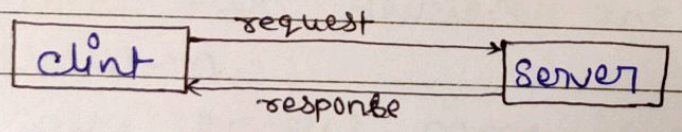
AWT	SWING
① Heavy weight	① Light weight
② uses system resources heavely	② uses less system resources
(3) less components - Bottom Input	(3) more components - Bottom, Inputfield, textfield, textarea, Password.
(4) No MVE	(4) follow MVE (model view controller)
(5) No External pluggings	(5) we can Implement pluggings.

(2) Web Applications :-

Ex → music player, antivirus etc.

"Application which runs on the browser and whoes execution takes places on the server."

It workes on the clint-server model Architecture.



* Technology Used :- servlet, JSP, Frameworks.

(3) Enterprise Application :-

Enterprise Applications are distributed in nature.

* Technology Used :- EJB → Enterprise Java beans.

(4) Mobile Application :- (1) J2ME (Java to micro Adelition)
(2) Android

(5) Games, Embedded system, Robotics etc.

*. Features of Java :-

- Simple and Easy to understand.
- Java is secure (friend function and pointer are absent thats why this is secure language).
- Java is a Portable Programming Language.
- Java is a platform Independent or m/c Independent language.
- Performance is good or moderate comperission to C and C++.
- Multitasking and Multithreading
- Robust (means having highest strength)
- It is based on OOPs.
- Automatic Memory Mangement (Garbage Collector) simply removes dereferenced.
- uni code → char → 2 (universal International standard character incoding set).

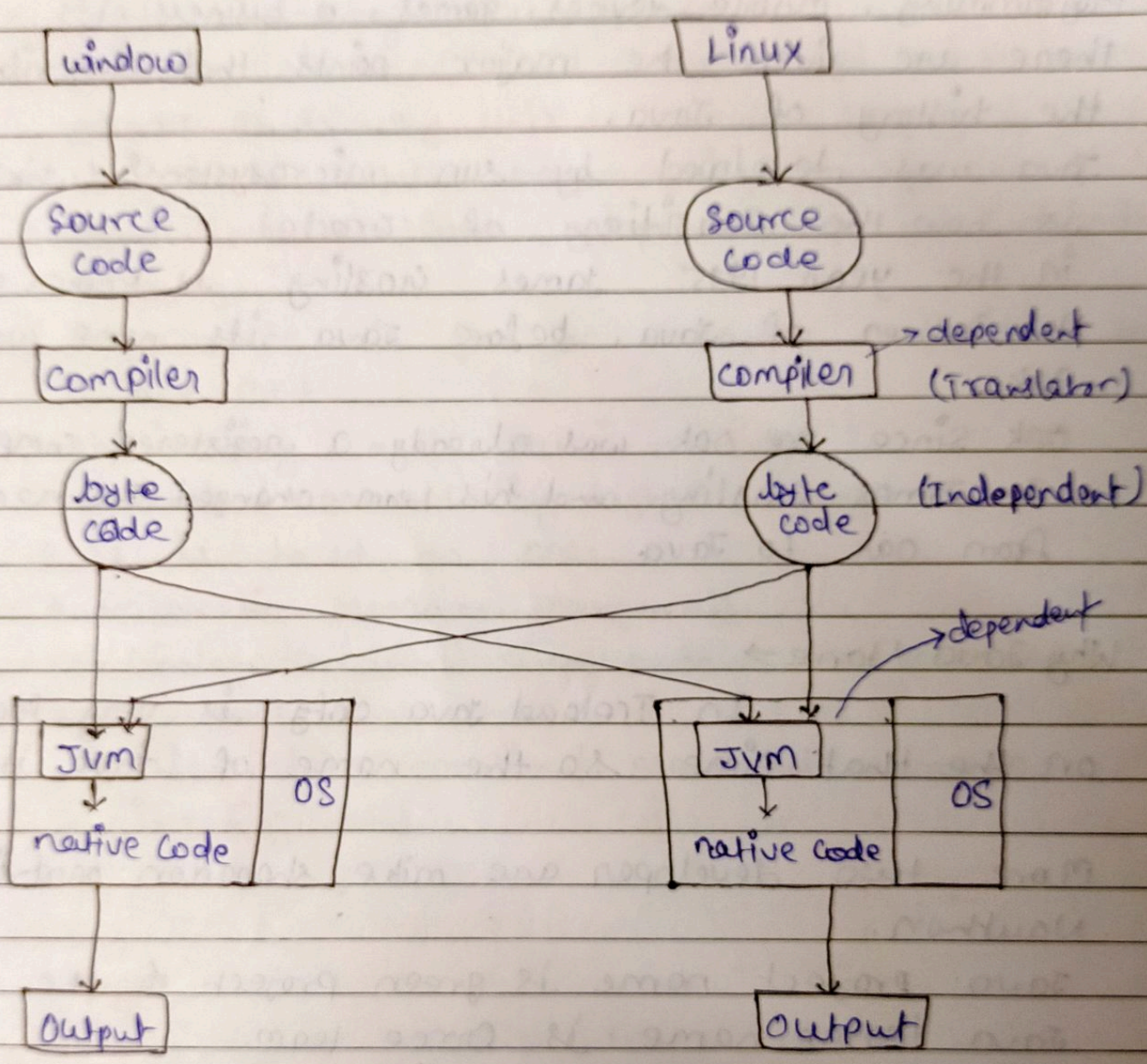
* History of Java :-

Currently, Java is used in internet programming, mobile devices, games, e-business etc.
There are given the major points that describes the history of Java.
Java was developed by sun microsystems (which is now the subsidiary of Oracle)
in the year 1995 James Gosling is known as the father of Java. before Java its name was Oak.
~~Oak~~ Since ~~Oak~~ Oak was already a registered company. So, James Gosling and his team changed the name from Oak to Java

Why Java Name →
In Iceland Java cofy is very famous on ~~the~~ that time so the name of java is java.

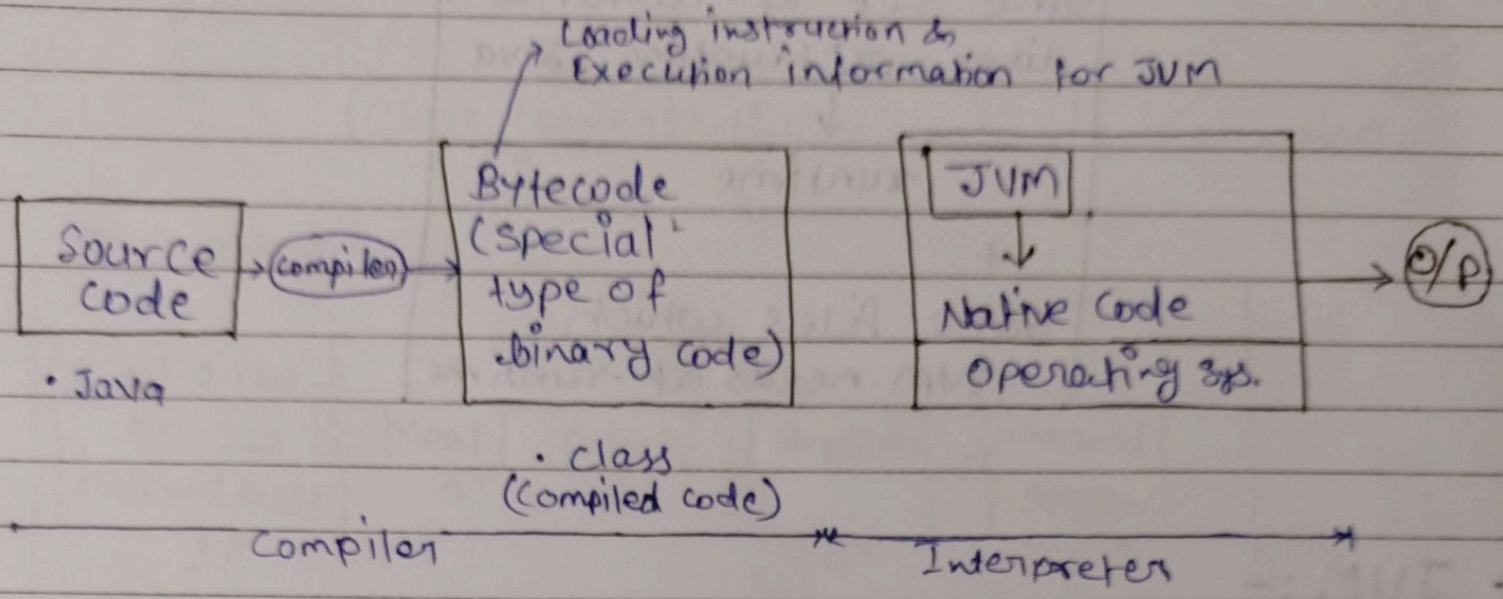
More two developer are mike sharadon, partric Nauthon.
Java project name is green project & the Java teame name is Oree team.

* Java Platform Independent :-

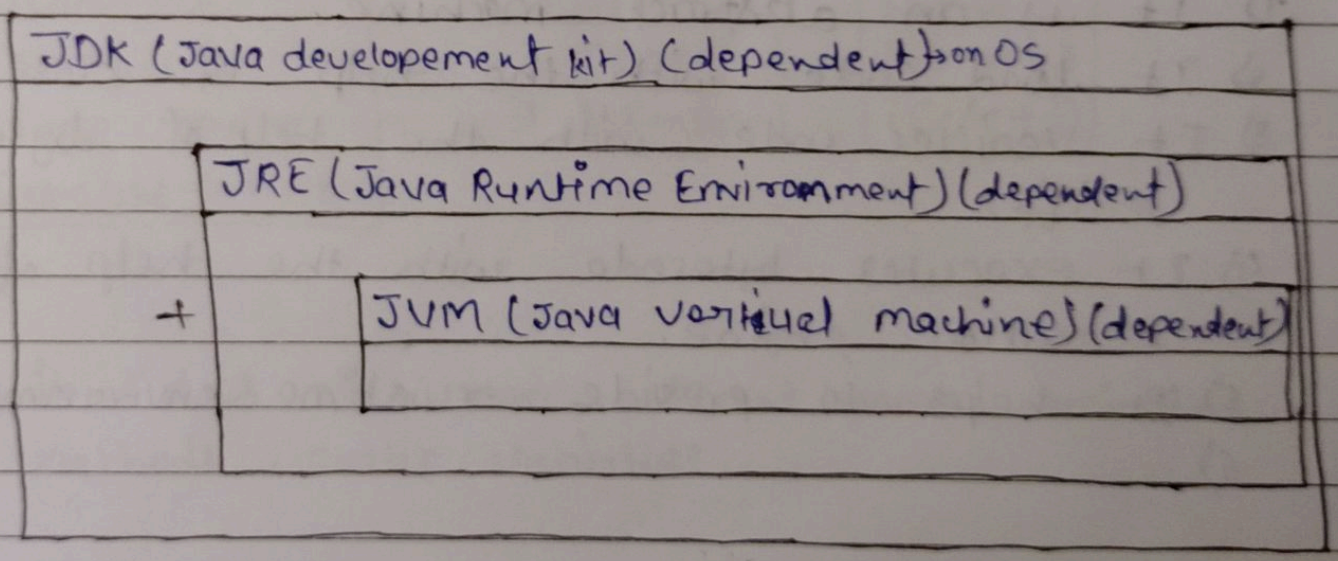


JVM = Java Virtual Machine.

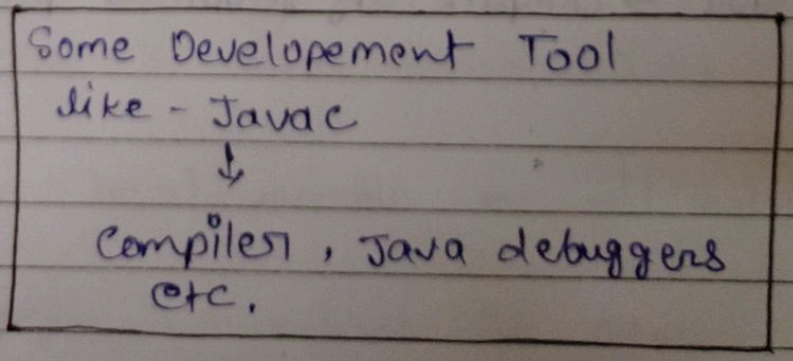
#. Flow of Java Program :-



*. JDK, JRE, JVM :-



In JDK :-



In JRE :-

Some set of libraries like rt (runtime).Java ↓ runtime
+ Some files which JVM needs at runtime

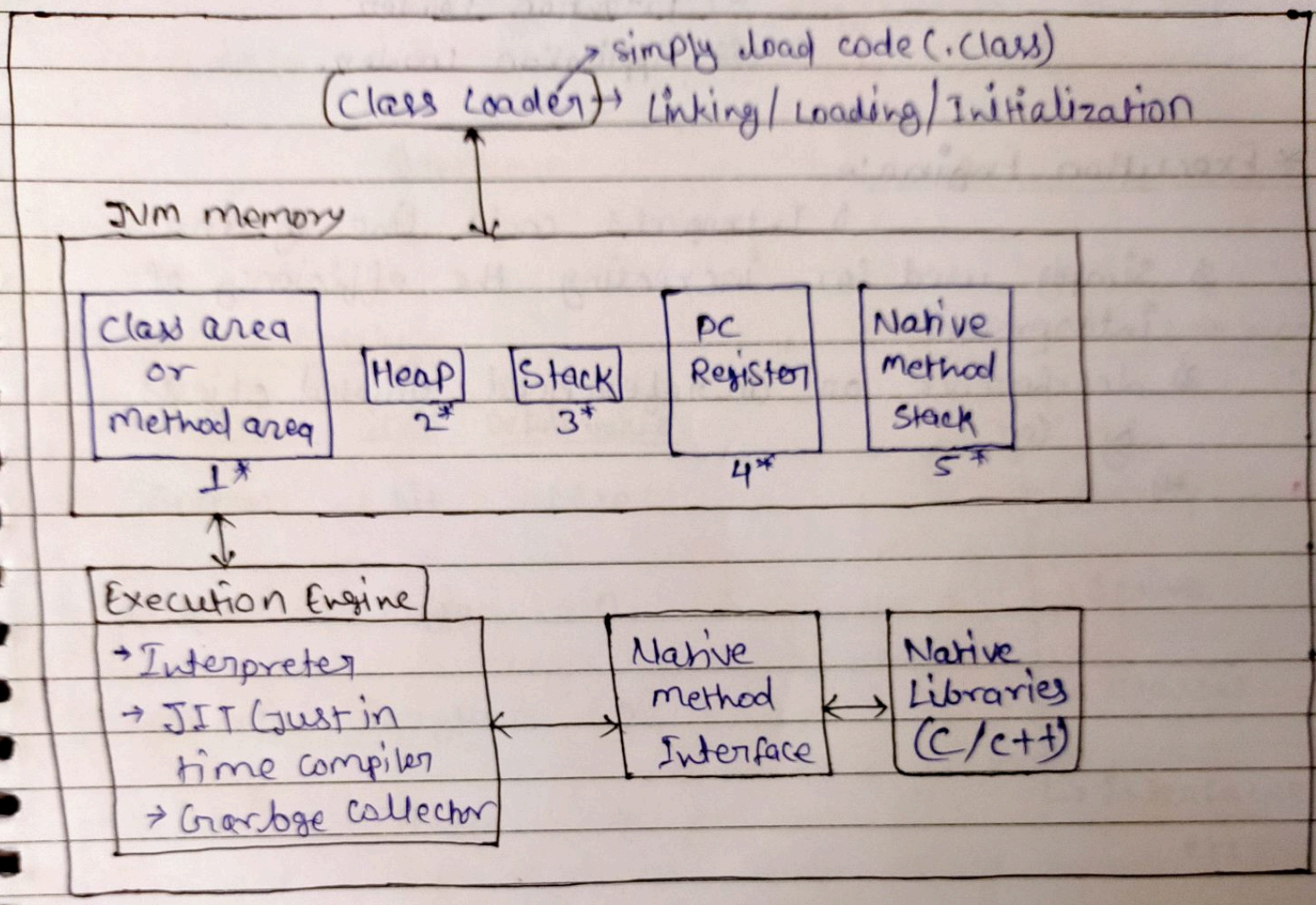
* JVM :-

(Java virtual Machine)

1) It is an abstract machine.
2) It load code with the help of class loader.
3) It verifies code with the help of bytecode verifier.
4) It executes bytecode with the help of execution engine.
5) It helps to provide runtime environment.
6)

*. program.java → compile by javac → program.class

* Architecture of JVM :-



1*) → it holds class name, Parent name, variables, Methods, static variables.

2*) → 1) Memory allocation throught new keyword.
2) Dynamic memory allocation.
3) from JDK & static variables falls in Heap.

3*) → It hold local variable as well method calls.

4*) → holds some Instructions for JVM

5*) → deal with thread.

* Classholder :- loading → (1) hold bootstrap loads.
(2) Extantion Loader
(3) Application Loader.

* Execution Engine :-
1) Interprets code line by line.
2) simply used for increasing the efficiency of interpretor.
3) dereference or unreferenced unused objects by GC.
4)

Data Type :-

Datatype simply represents the type of data and size of Data.

Datatype

- Premitive Datatype
- Non-Premitive Data type.

Datatype	Size	Defalt value
(1) Boolean	1 bit	false
(2) byte	1 byte	0
(3) Short	2 byte	0
(4) int	4 byte	0
(5) long	8 byte	0
(6) float	4 byte	0.0
(7) double	2 byte	0.0
(8) char	2 byte	\u 0000

Non-Premitive Data type.

- Array
- String
- Classes
- Interfaces

etc.

* defaut value of Non-premitive datatype (Non-premitive reference variable) is NULL.

* Java Programming structure :-

```
class <class-name>
{
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}
```

* class Hello → class name

keyword

→ user defined blueprint for creating object

→ By conversion, first letter of class name should be in uppercase.

* When we use pubic access modifier before main 'class name' so at that place our class name and file name is same otherwise different.

Ex → if

```
public class Hello{

    }
```

here file name must be Hello.java.

* public class Hello {
Here in this case, its mandatory to have class name and file name same.
and here public is an Access modifier.

* Access Modifiers – (1) public
(2) private
(3) protected
(4) default (Package private)

* { → Opening of class definition part

* public static void main (String args[ ])
Here Public → · Access modifier
· represents Accessibility
· JVM can call It from any where
static → If we make any method as a static, then In that case, we not need class object to call that static method. it is directly called by class name.
Static method can be overloded where as static method can not be overriden.
Static is a modifier. (like – abstract, final, strictfp synchronized etc.)
void → return type (which does not return any value).
main → predefined name (function name).
(String args[ ]) → command line
It's responsible for taking arguments for your main method at runtime.

String → Predefined Class
args → String array name
[ ] → square brackets.

→ public static void main (String args[])
→ public static void main (String [] args)
→ pubic static void main (String[] args)
→ public static void main (String ... args)
→ ↓ varargruments
→ public static final void main (String args[])
→ final public static void main (String args[])
→ static public void main (Strings args[])
→ final strict fp public static void main (String args[])
↓ strict floating point

* String (S), (args [])
↓ variable ↓ array

String (S), ([] args)

String [] (S, args)
↓ array

→ { → opening of main method definition part

(System) (.) (out) . (println) ("(Hello)");
→ string
→ new line

↓ class ↓ dot operator ↓ object of → ↓ method of printstream class

} → closing of main method.

* class System (static member of System class)

```
{
    Static printStream out;
}
```

printStream → class; out → object of

out is a object of PrintStream class

* Class PrintStream

```
{
    pubic void println (String s){
    }
    Public void print (String s){
    }
    public void printf (String s){
    }
    public void println (int i){
    }
}
```

* Program in Java :-

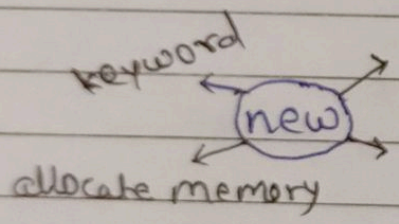
```
Class Demo
{
    public static void main (String args[])
    {
        int a= 100;
        System.out.println ("Value of a is : "+a);
    }
}
```

* In Java User Define (value getting):-

nextInt()	int
nextFloat()	float
nextDouble()	double
nextLong()	long
nextShort()	short
nextByte()	byte
nextBoolean()	boolean
nex()	single word] string
nextLine()	Sentencod
next().charAt(0)	character

→ Scanner (class)

*



*

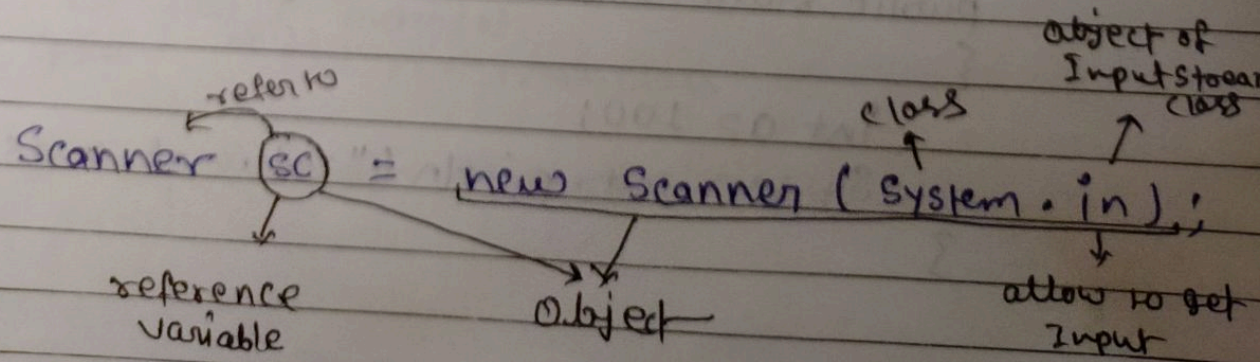
new Scanner ( ) ;

↗ Here we pass objet

↓

object
(anonymous object)

*



*

```
class System
{
    static InputStream in;
}
```

in is static member of System class.

InputStream → class

in → object of InputStream

*

```
int a = sc.nextInt();
~~int~~ float b = sc.nextFloat();
char c = sc.next().charAt(0);
```

*

```
class Scanner
{
    int nextInt() {
        return statement
    }
    float nexfloat(){
        return stmt
    }
    char next.Cho
        return stmt
    }
}
```

*

```
import java.util.Scanner;
```

Scanner → class.

java → Package

util → sub-package

*Controll Statement :-

the statement or Instructions are used to inform the Computer to perform the task when the program is being executed.

there are 4 types of control statements -

(1) Sequential control statement
(2) Decision control statement (conditional control stmt)
(3) Switch control statement
(4) Looping control statement

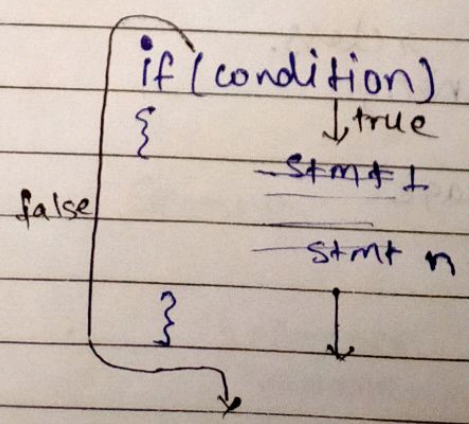
*(1) Sequential control Stement →

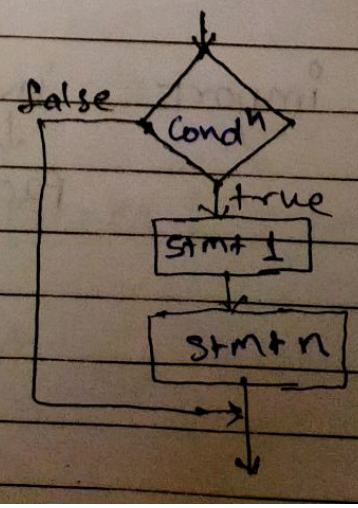
Action1 Input → Action Process → Action3 Output

Ex.→ I/P P, r, t SI = (p*r*t)/100 O/P SI

*(2) Decision Control Statement →

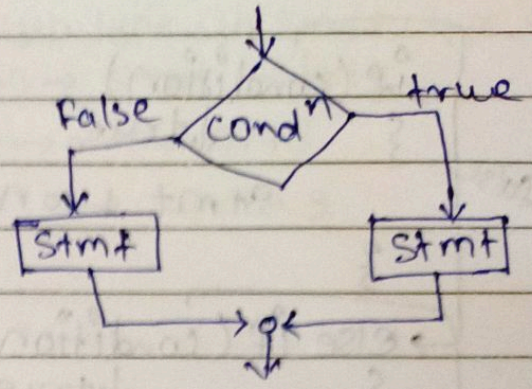
*(i) if statement





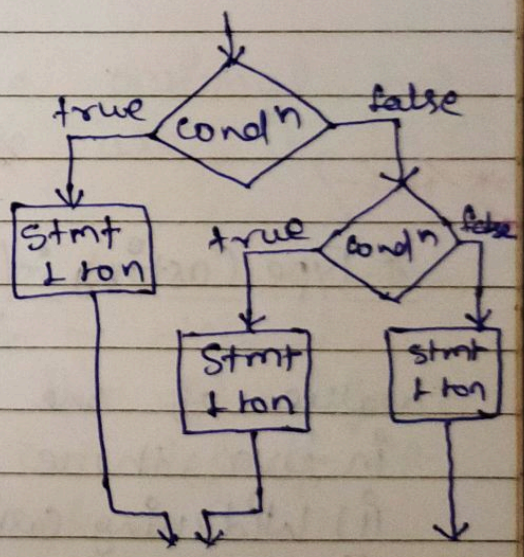
(ii) if else Statement

```
if (condn)
{
    Statement 1-n
}
else
{
    Stmt 1-n
}
```



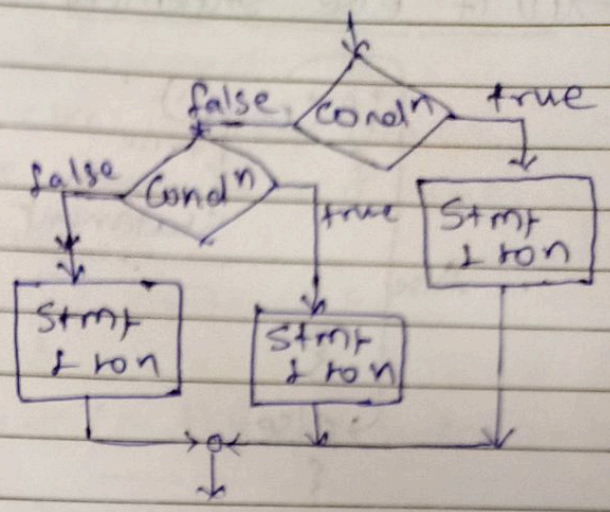
(iii) Nested if else Statement

```
if (condition)
{
    Stmt 1 to n
}
else
{
    if (condition) {
        Stmt 1 to n
    }
    else {
        Stmt 1 to n
    }
}
```



(iv) Else if Ladder :-

```
if (condition)
{                ↓true
        stmt 1 ton
}
else if (condition)
{                ↓true
        stmt 1 ton
}
else*
{
        stmt 1 ton
}
```



* else is not compulsory.

* NOTE - typecasting → (1) Premitive typecasting
(2) Reference typecasting.

* Type Casting :- (Premetive) :-

Type casting is when you assign the value of one premitive data type to another premitive type.

in java there are two type of premitive type casting -

(i) Widening casting (automatically)

(ii) Narrow casting (manually)

(Implicit)

(i) Widening Casting :- (automatically) (data is safe)

Converting a smaller type to larger type size.

byte → short → char → int → long → float → double

Ex→ int num = 9;

double newnum = num; → output → 9.0

data safe → int 4 byte → float 4 byte → double 8 byte

data loss:-

double 8 byte → float 4 byte

here 4 byte can be lose

(Explicit)

(ii) Narrowing Casting :- (Manually) (data is not safe)

converting a larger type to smaller type size.

double → float → long → int → char → short → byte

Ex→ double myDouble = 9.78d; → output 9.78

int myInt = (int) myDouble; → output → 9

(3)

Switch Control Statement :-

① switch control stmt is mainly used for ~~menu~~ to menu based programming

2> it works on the concept of cases.

3> duplicate cases are not allowed.

4) the selection of cases takes place by constant, variable and expressions.

5) here **brake** and **default** are optional. but its a good practice if we use it in switch control stmt.

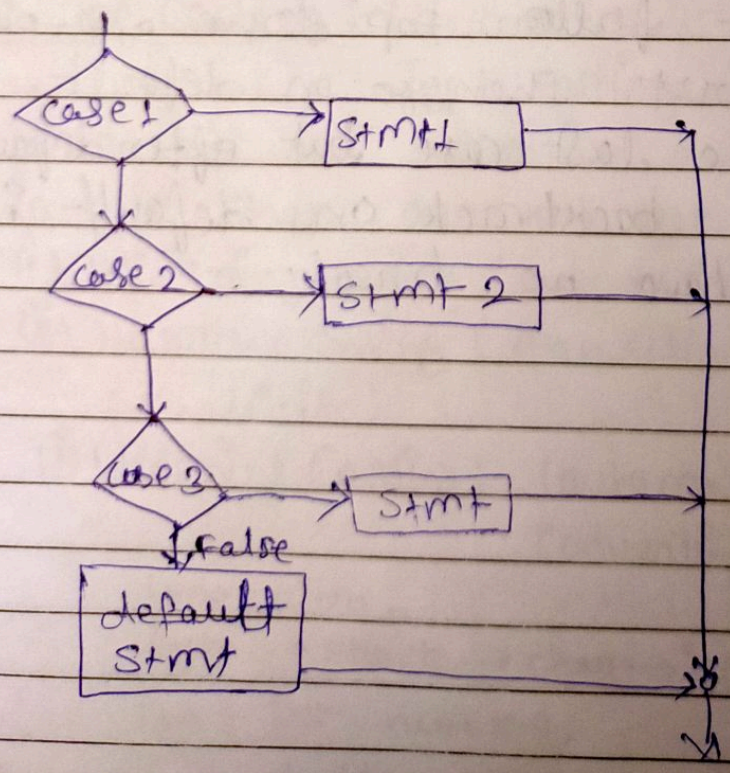
6) Switch case does'nt follow top down approach. it chekes cases first then go on default. if case match in the last case but after default then it will never backtrack on default if matched case ~~have~~ have no break;

*Syntax:-

```
switch(constant/variable/expression)
{
    case 1:
            brake;
    case 2:
            brake;
    case 3:
            break;
    default:
            break;
}
```

```
case 'a':
        break;
case 'b':
        break;
case 'c':
        break;
default:
        break;
```

*Flowchart:-



*. Inside Doubts :-

Source Code (English like Statement) $\xrightarrow{\text{Javac}}$ .class file (bytecode / visible / specific type of binary code)

$\downarrow$ JVM Interpreter

Binary code (native code) (readable for CPU) $\longrightarrow$ Executed by CPU

*. Binary language is platform dependent. It means different for different CPU's.

*. bytecode is independent that's java .class file is independent.

(4) # Loop Control Statement :-

Looping is the process of performing same task or some task in an iterative manner until certain condition is ~~sto~~ satisfied.

there are ~~three~~ types of loops
(1) Entry controll loop
(2) Exit control loop
(3) Inhenced ~~top~~ loop or Advanced loop or foreach loop.

(1) Entry control loop :-

In case of entry control loop condition is checked at the time of entry.
for loop and while loop falls under this Category.

(2) Exit control loop :-

In case of exit control loop condition is checked at the time of exit.
do while loop falls under this category.

(3) foreach loop :-

It acts as a container loop. this type of loop only deals with array and collections.

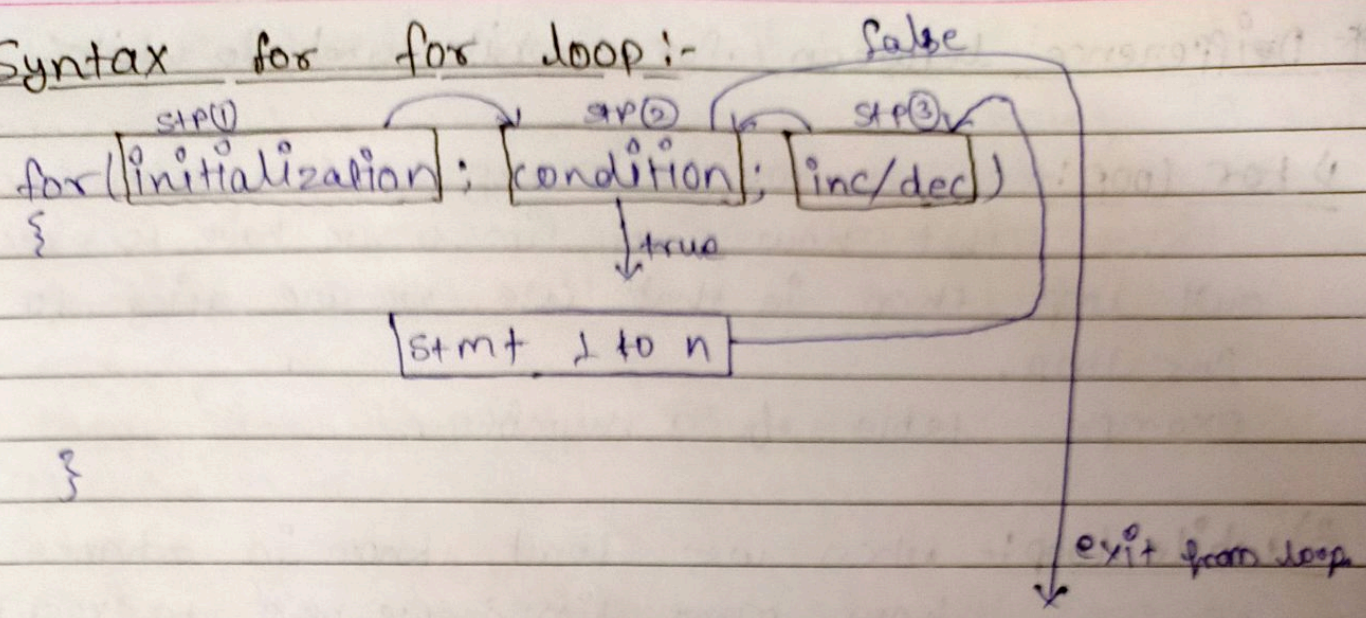
* Difference between for, while and do-while:-

1) **for loop:-** when we allready know in advance that how many times we have to execute our loop then in that case we are going to use for loop.
example - table of a number.

2) **while loop:-** when we don't know in advance that how many times we have to execute our loop then in that case we are going to use while loop.
example reversing a number.

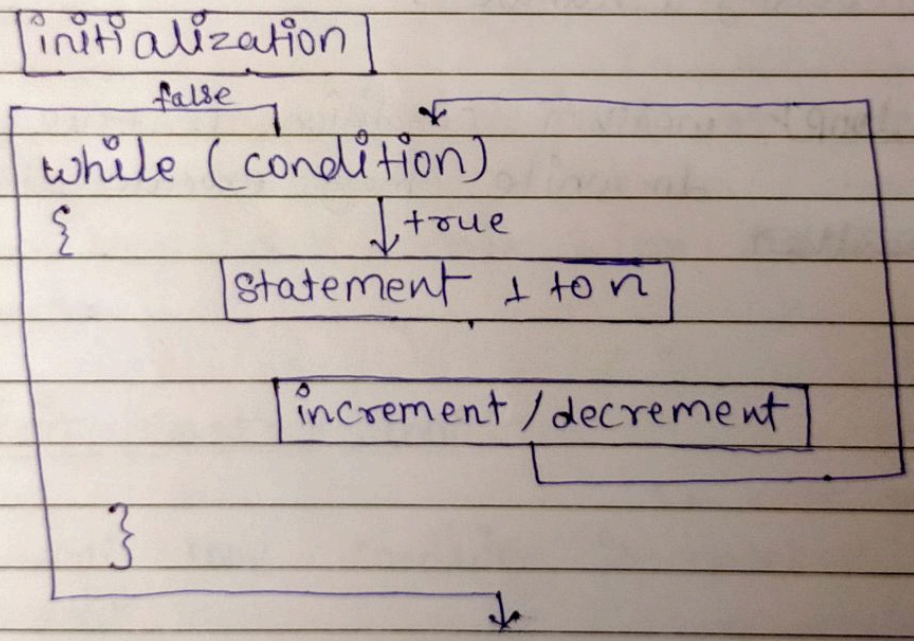
3) **do-while loop:-** weather condition is true or false do-while may executes at-least once. ~~weather~~

* Syntax for for loop :-

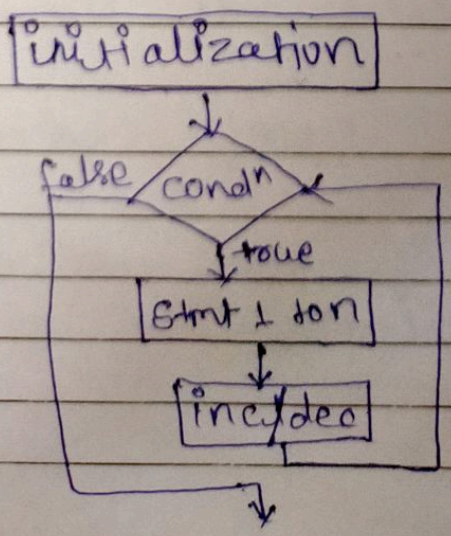


* Syntax for while loop :-

syntax



flowchart



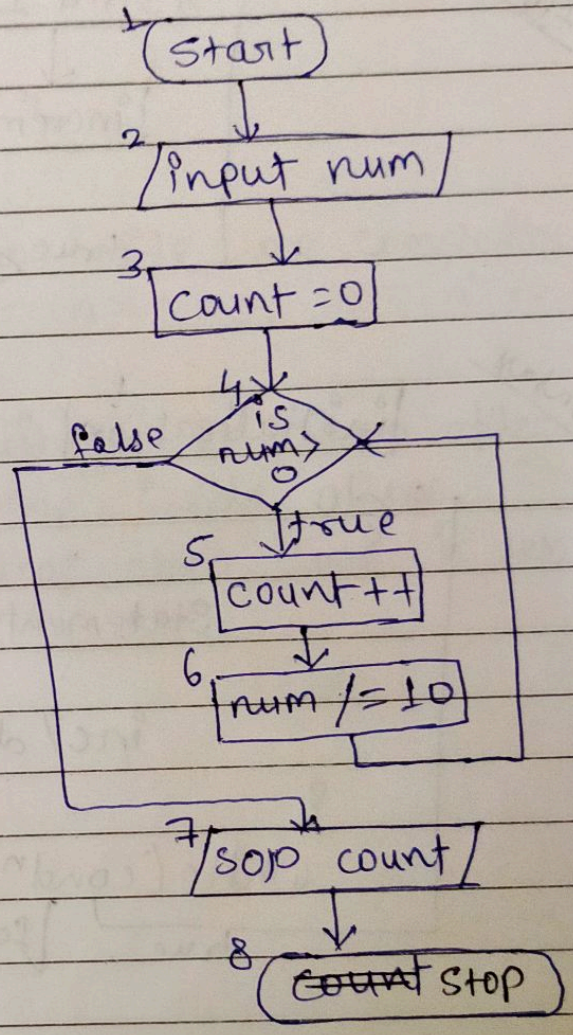
* Example of While loop :-

Problem - cont the no. of digits in a number.

Program -

```
num = input , count = 0
while ( num > 0 )
 {
        count ++;
        num = num / 10;
 }
 sop ( count )
```

Flowchart

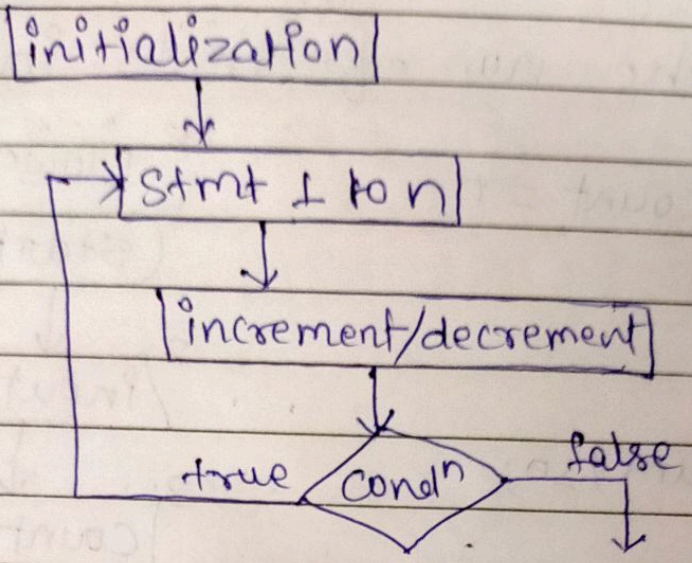


Algorithm -

Step ① - Start
Step ② - input num
Step ③ - initialize count = 0
Step ④ - if num > 0 then go to Step ⑤ otherwise go to Step ⑦
Step ⑤ - calculate ~~ct~~ count ++
Step ⑥ - calculate num /= 10
Step ⑦ - sop count
Step ⑧ - Stop.

* Syntax for do while loop :-

~~Syntax~~ flowchart



Syntax

```
initialization
do
{
        Statement 1 to n ;

        inc/dec ;
}
while(condn);
```

true false

* Nested Loop :- Loop inside loop.

①
```
for( ———— ){

}
for( ———— ){

}
```

(2)
```
for( ———— ){
    for( ———— ){
    }
}
```

(→) loop after loop complexity
$= n + n + \ldots$

(→) loop inside loop complexity
$= n \times n \times \ldots = n^2 \times \ldots$

* If task is successfully compliting with loop after loop then don't use loop inside loop. because the complexity of loop after loop is less then loop inside loop.

* Patterns :-

Array

M

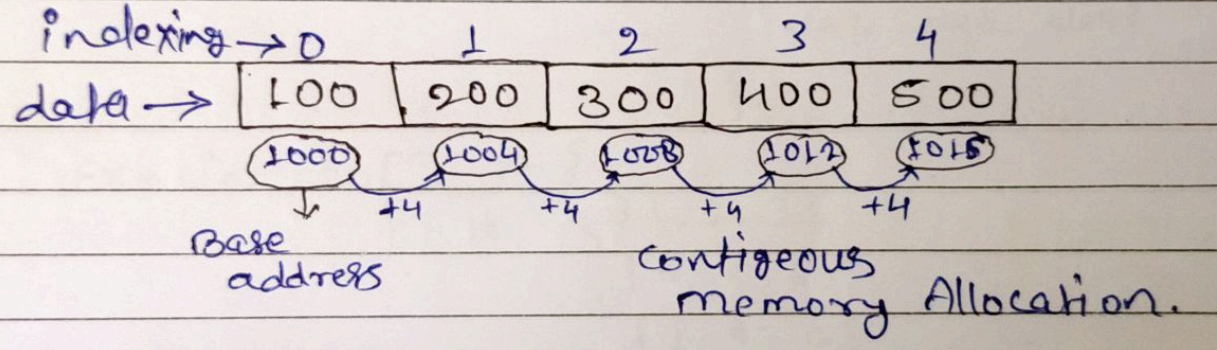
- → Array is a non-premetive datatype (refference type).
- → Array is a collection of Homogeneous data elements (same type).
- → Array works on the concept of indexing
- → Array Indexing always starts from 0 because it follows number system.
- → pre-requisite for array allowcation is base address.
- → Array works on contigeous memory allocation.
- → Array elements shares same name but differentiable by there indexing.
- → Manupulation ~~is~~ in Array is slow because lots of sifting is required when we insert or delete any element.
- → Array is a object in java.
- → Array is dynamically Allocated.
- → Once we define the size of Array then we can't change it.
- → Direct parrent class for array type is object class.
- → Array can hold premetive data types as well as object refference.
- → length of array = arr.length
  = <array_name>.length

Types of Array:

there are three types of Array:

1) 1 - Dimmentional Array
2) 2 - D Array (Jagged Array is Special kind of 2-D Array)
3) Multi Dimmentional Array

* 1- Dimentional Array :-



Syntax:

1) <datatype><array_name>[] = <datatype>[] { ele1, ele2, ele3, ... elen };

Ex→
int arr[] = new int[]{10, 20, 30, 40, 50};

2) <datatype><array_name>[] = new<datatype>[size]

Ex→
int arr[] = new int[5]
for (——) {
——— } insertion value.
——
}

3) <datatype> <array name> [ ] = { ele1, ele2, --- elen };

ex→ int arr[] = {10, 20, 30, 40, 50}.

* 2 Dimentional Array :-

→ 2D Aarray is a collection of 1D Aarray

→ it works on grid system. (rows & colums.).

Syntax: (1)

```
<datatype><array_name>[][]= new<datatype>[row][col];
```

Ex→ int arr[][] = new int[2][3];

(2)
```
<datatype><arr-name>[][]= {
                    { ele1, ele2, ele3 },
                    { ele4, ele5, ele6 },
                    { ele7, ele8, ele9 }
                    }
```

Ex→
```
int arr[][] = {
              {1, 2, 3},
              {4, 5, 6},
              {1, 9, 0}
              }
```

→ length of 2D Aarray (not jadeed) :-

arr.length * arr[0].length.

→ In the case of memory management foreach loop is not good because it creates new array to extract the elements ~~which~~ that why foreach loop is not good compare to other loops.

* Jagged Array :-

→ Jagged array is a special type of 2D array.

→ rows → fixed

columns → Different

row ↓ / column →	0	1	2	3	4
arr[0] 0	18	20	21	13	25
arr[1] 1	19				
arr[2] 2	24	23	1	2	
arr[3] 3	6	5	9		

= new int

Syntax :-

<datatype> <array name> [ ] [ ] = new <datatype> [row] [ ];

Ex→ int arr[ ][ ] = new int [4] [ ];

arr.length = 4

arr[0] = new int [5] = new int[]{18,20,21,13,25};

arr[1] = new int [1] = new int [] {19};

arr[2] = new int [4] = new int[]{24,23,1,2};

arr[3] = new int [3] = new int[]{6,5,9};

→ the range and size of premitive datatypes is depends on System Architecture.

*

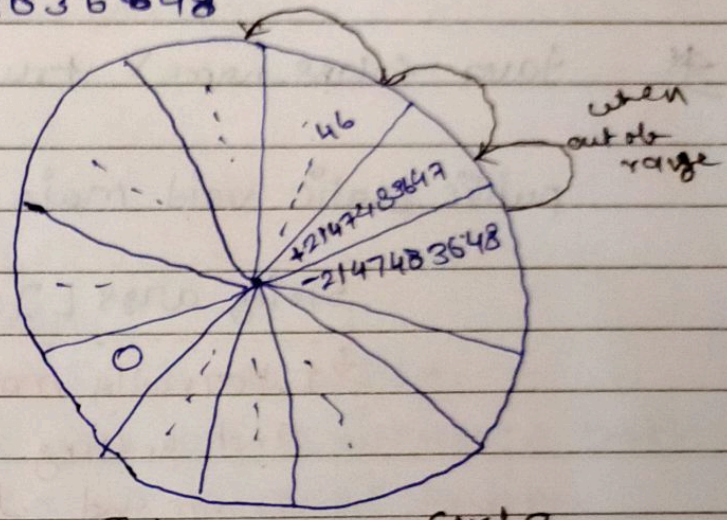
Out of Range Concept -

→ SOP(2147483647 + 3) → O/P → -2147483646

highest Integer → 2147483647 + 3;

Lowest Integer → -2147483648

→ this concept is applied for all premitive data types.



Integer range cycle

#. Command Line Arguments :-

Publi static void main(String args[])

↓

Command Line Arguments

→ Responsible for tacking arguments for main method at runtime.

javac <file_name>.java

java <class_name> <arg1> <arg2> .....

↓ command (program .exe)

Line

↓ arguments.

→ Arguments are always Input as a String.

Ex →

<arg1> → 1 2 3

<arg2> → Hellow user how are you.

<arg3> → true false.

* java <class_name> true false ...

public static void main (~~arg~~String args [])

```
String args [] = {"true", false"}
```

↓ Internally making an string array and getting ~~comm~~ Arguments of Command line

SOP (~~arg~~ args [0]); → O/P → true.

↓ String datatype

}

* Paradigm → single style of Programming (oops) (java).
* Multiparadigm → multiple programming style (oops + pop) (C++) (Python)

object oriented programming System (style).

OOP's Concept :-

(1) Object
(2) class
(3) Inheritence
(4) Abstraction
(5) Polymorphism
(6) Encapsulation

(1) Object :-

→ Object is a Real world Entity.
→ It is also termed as Runtime entity.
→ Every object have some physical Existence that's why we can call it as a physical Entity.
→ Every object belongs to some specific class.
→ object is an instance of a class.
→ objects may communicate with each other with the help of functions.
→ every object have some state, Identity and behaviour.
*→ Runtime entity means the data members and member functions are travel along with the object.

(2) Class :-

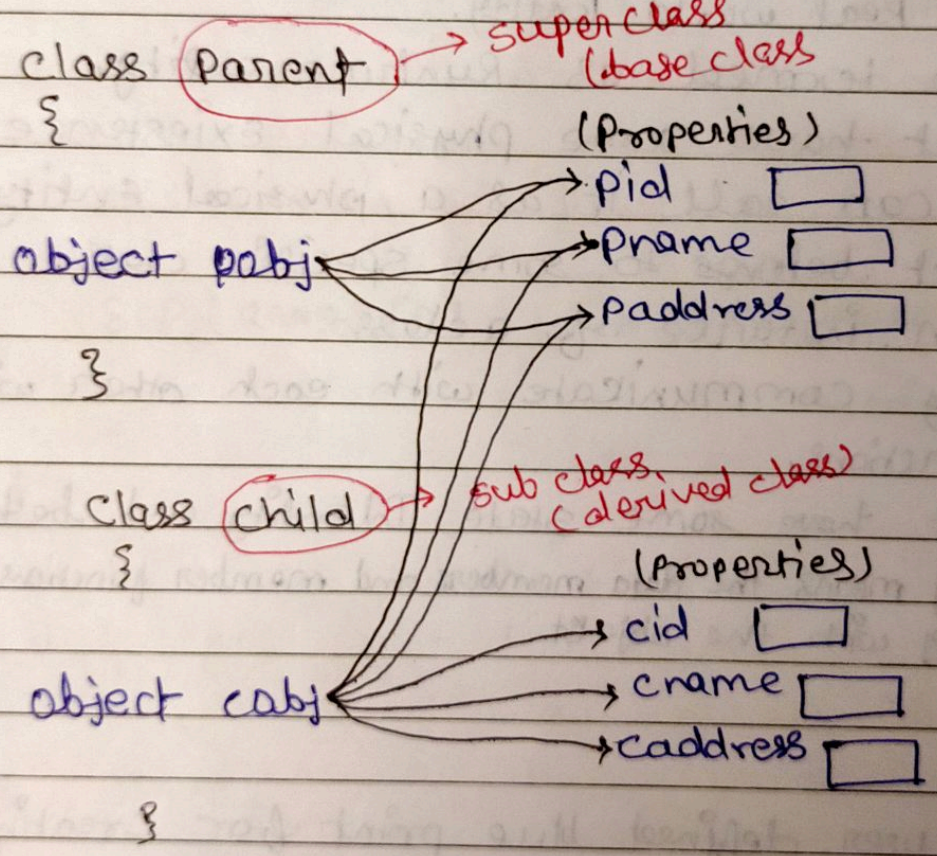
→ Class is a user defined blue print for creating object.
→ class is the collection of similler type of objects.
→ class is a logical entity.
→ Class provides specification
→ Class consist of data members and member functions.

(3) Inheritance :-

→ Inheritance is the process in which object of one class may inherit the properties of object of another class.

→ It works on the concept of code re-usability.

→ here parent, child relationship takes place. where parent class is also known as superclass or Base class and child class is also known as Subclass or Derived class.



* Types of Inheritence :-

there are three types of Inheritence:

1. Single Inheritence
2. Multilevel Inheritence
3. Hierarchical Inheritence
4. Hybrid Inheritence

(4) Abstraction :-

→ abstraction is the process hiding implementation details and showing only functanility to the users.

→ It works on the concept of data hiding.

→ here we use the concept of abstract class to implement the functanility of abstraction on multiple classes.

→ Using the things without knowing its background details (Processing) is known as abstraction.

(5) Polymorphism :-

→ polymorphism is derived from a greek term where 'poly' means 'many' and 'morphism (morph's)' means 'forms' which means "more then one form".

→ If we want to achieve the functanility of abstraction in single class then in that case we are going to use the concept of Encapsulation.

Polymorphism can be classified as —

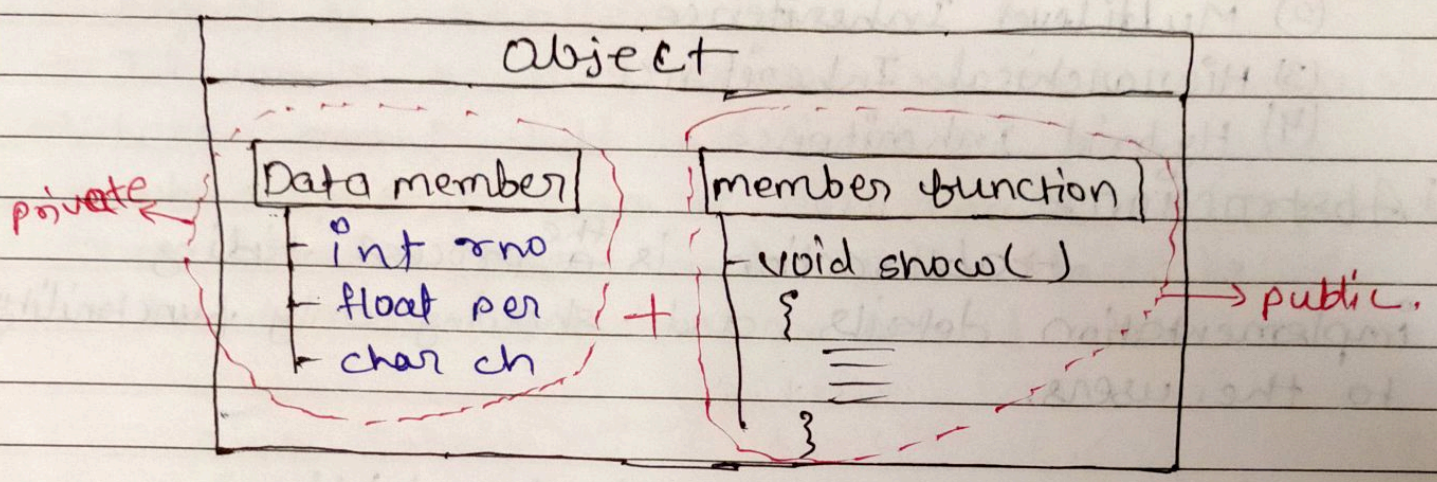
(i) Compile time polymorphism
OR static Binding
OR Early Binding

(ii) Runtime polymorphism
OR Dynamic Binding
OR Late Binding.

* Same name different work.

(6) Encapsulation :-

→ ~~Bo~~ wrapping up of data members in member functions into single unit that is an object is known as Encapsulation.



→ for acheiving encapsulation in more efficient ~~maner~~ manner we need to make our data members as a private and member functions as a public.

* Thumbrule of OOP'S :-

object wich you want to manupulate first think about that then identify the properties of the object then identify the behaviour of object then finally create the factory (class).

* Difference between Abstraction and Encapsulation :-

(1) Abstraction always achieves at the user level. vs
Encapsulation always achieves at the developer level.

(2) Abstraction is user level data hiding. vs
Encapsulation is developer level data hiding.

Function :-

function is a self contained block of statements.
Every function performs some specific task
there two types of functions -
(i) Predefined function
OR Inbuilt function
OR Builtin function
OR Standard function
OR Library function
(ii) user defined function

(i) Predefined functions :-

→ this are the functions whose defination part is already defined.
Ex→ println( ) ; print( ) ;

(ii) user defined functions :-

→ this are the functions whose definition part is defined by the user or programmer according to programming needs.

```
Ex→ void show(){
      ___
      ___
    }

int main(){
  ___
  ___
  return 0;
}
```

* Advantages of function :-

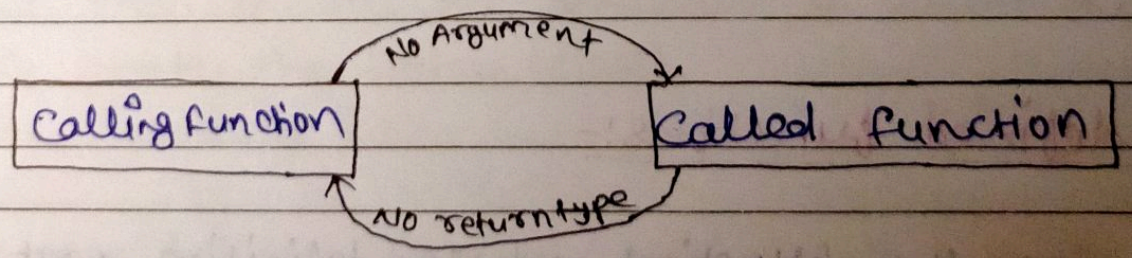
→ function reduces program size (no. of lines).
→ Function Increases code reusability.
→ function reduces redundency of code.
→ function reduces complexity.
→ debugging would be easier.
→ we can create n number of function.
→ a function ~~cat~~ can be called n number of times.
→ functions can not return more then one value at a time.
→ void represents no return type (value).

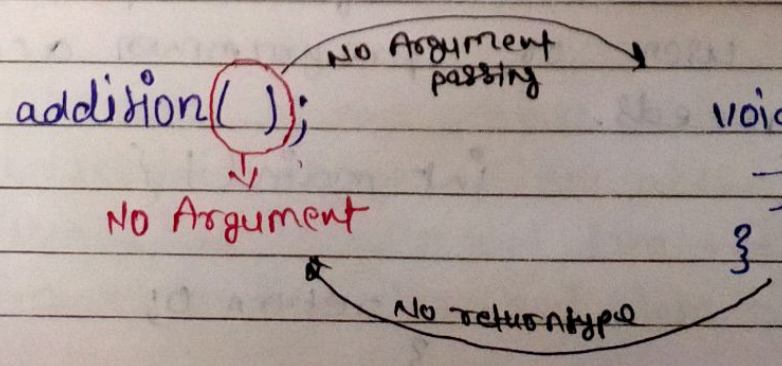
* Categories of functions:-

there are 4 categries of functions -

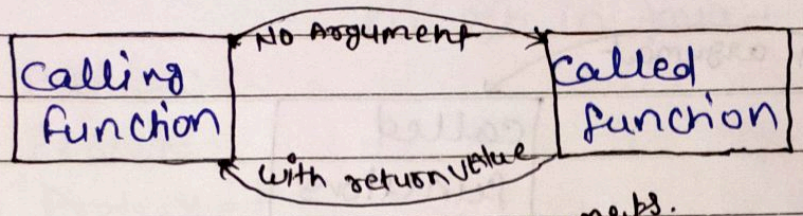
(i) No argument No return type
(ii) No argument with return type
(iii) with argument with return type
(iv) with argument No return type.

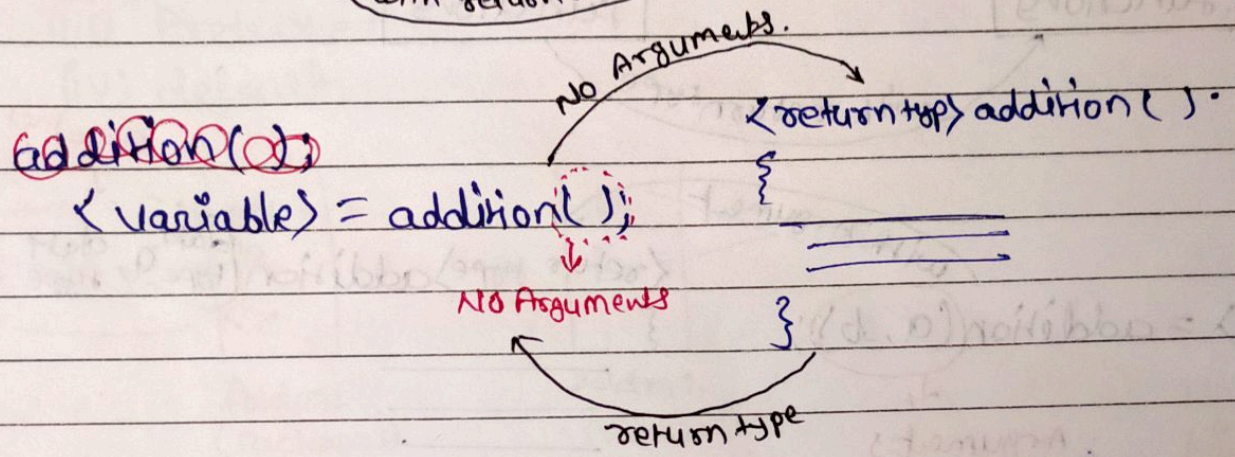
(i) No Argument No return type :-



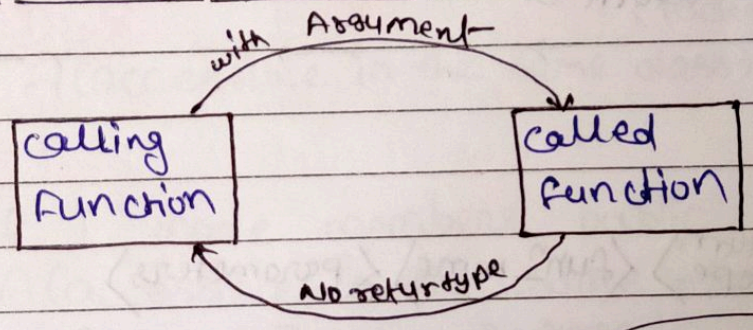


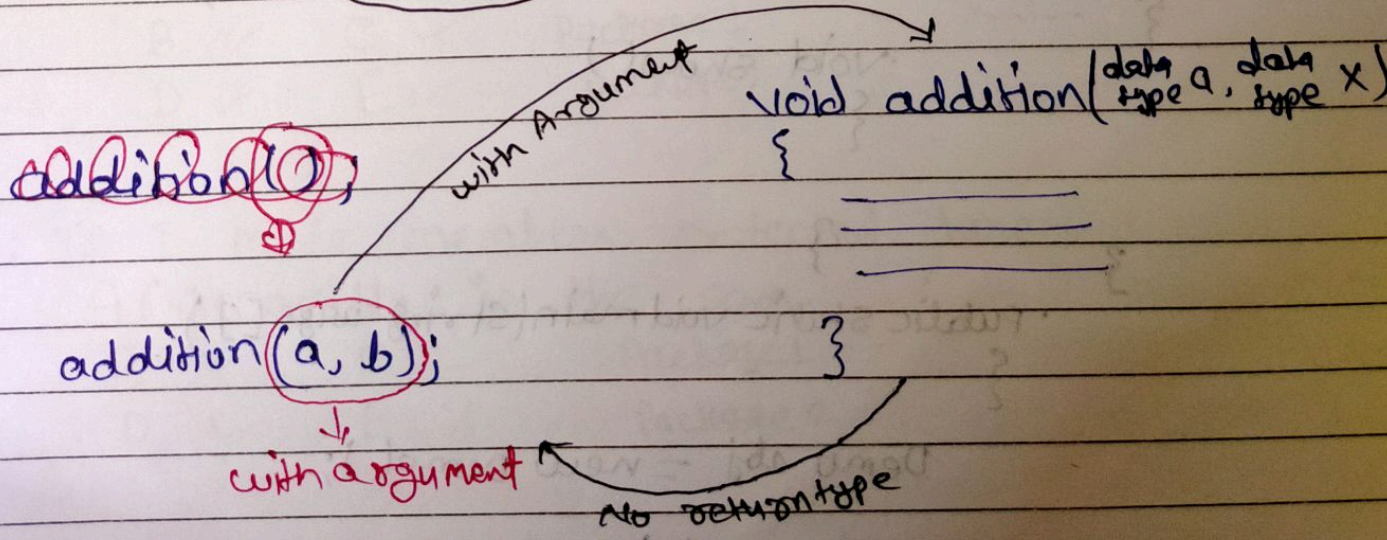
(ii) No Argument with Return type :-



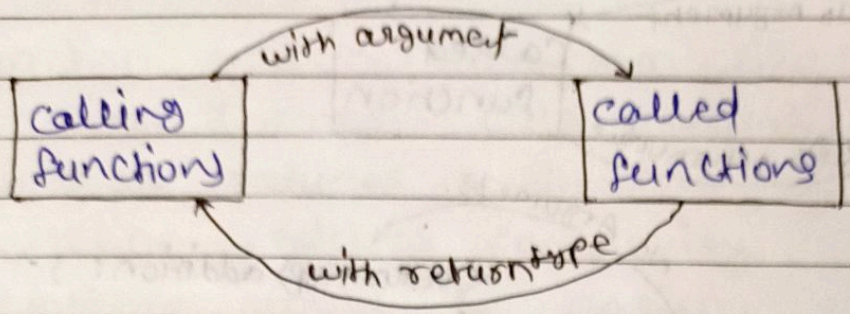


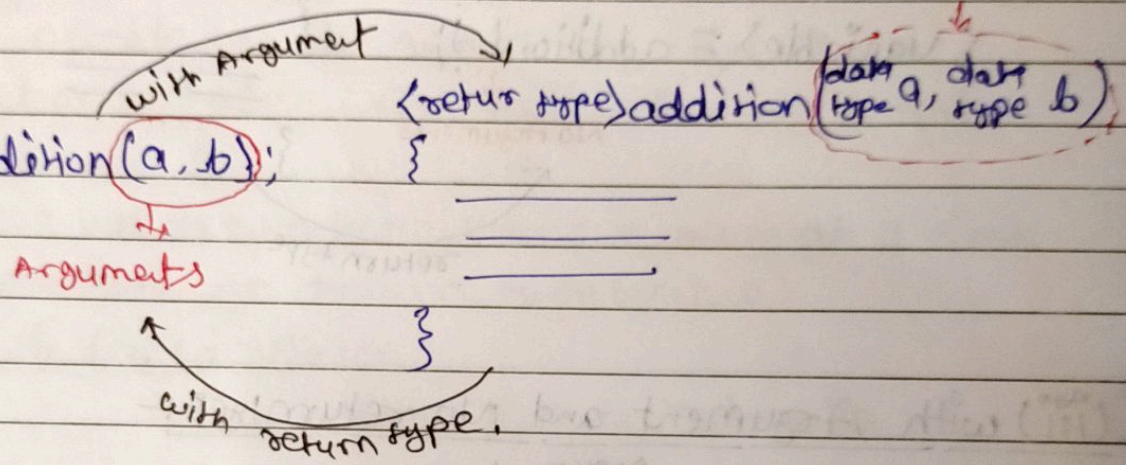
(iii) with Argument and No return type :-





(iv) with Argument and with Return type:-



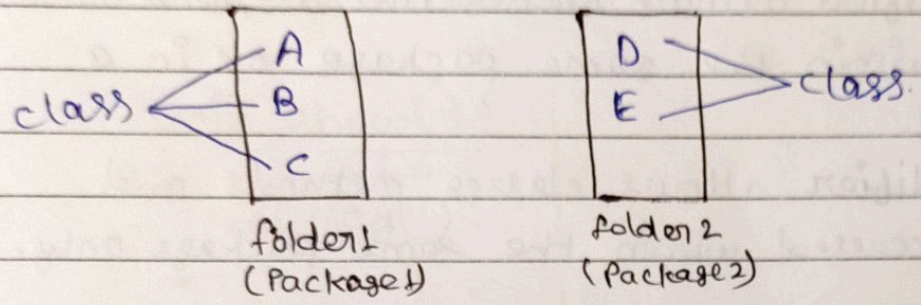



```
Class Demo
{
    <access modifier> <return type> <funⁿ name> <parameters>
    {
        void show()
        {

        }
    }
    public static void main(String args[])
    {
        Demo obj = new Demo();
        obj.show();
    }
}
```

* **Access Modifiers:-** there are 4 types of access modifiers in java-

(i) Public
(ii) Private
(iii) Protected
(iv) default



A: If I made members private here-

B ✗ C ✗ folder 1 (Package 1)
D ✗ E ✗ (Package 2)
A (accessible in the same ~~class~~)

A: If I made members public here-

A (accessible in the same ~~class~~)
B ✓ C ✓ Package 1
D ✓ E ✓ Package 2

A: If I made members protected here-

A (accessible in the sam
B ✓ C ✓ Package 1
D ✓ E ✗ Package 2

B: If I made members default here-

B ✓ C ✓
D ✗ E ✗

→ Access Modifiers are used to set the accessibility of classes, methods, and variables.

→ Public Access modifier allows classes, methods and variables to be accessed from any part of the code.

→ Private Access modifier allows classes, methods and variables to be accessed only within the same class.

→ Protected Access modifiers allows classes, methods, and variables to be accessed within the same package or in a subclass.

→ Default Access modifier allows classes, methods, and variables to be accessed within the same package only.

* Default = PKG Private
* the accessibility of default and protected are same but when child class is present in other package then the accessibility of protected is more.

* Recursion/Recursive Function:-

→ Recursion is the process in which a function calls itself in a repeatative manner until certain condition is ~~s~~ satisfied.

→ types of Rucursion-

(i) Direct Recursion

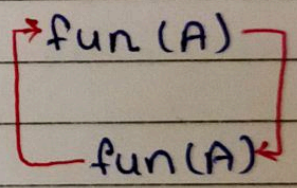
(ii) Indirect Recursion.

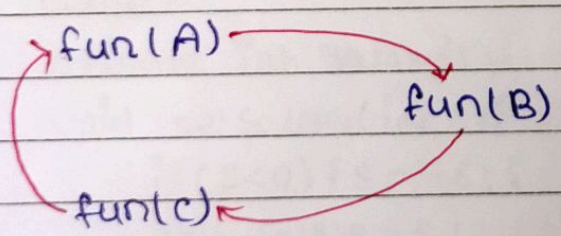
```
void show(){
    display();
}
void display(){
    show();
}
```

Indirect Recursion

```
void show(){
    show();
}
```

Indirect Recursion.





→ Tail Recursion : The recursive is the last statement.

→ Head Recursion: The recursive call is made at the beginning of the method.

→ Multi ~~Re~~ Recursion: Multiple Recursive calls are made in a method.

→ Mutual Recursion:

→ Nested Recursion:

→ Tower of Hanoi :

#1

Some chhotu Motu Topics :-

* hashcode → Default String Representaion of an object. Ex → class@12E43
→ driver class - class that contains main method
→ Error always occurs at compile time.
→ Exception always occurs at Runtime.

~~Data transfer object (DTO)~~
~~Sop (obj name);~~
~~here obj is dto because we setting the data~~
~~with the help of obj.~~

wrong stmt

~~→ DTO~~

→ JLS → Java Language Specification.

#. Getter & Setter Method :-

we are not able to access private data members outside the class but if we want to access this private members outside the class ~~then~~ in that case we use getter and Setter Methods

getter methods are used for getting the values and setter methods are used for setting up values.

*→ In general way - data members are private and member functions are public.

*→ Good Practice - for every instance variable make individually getter and setter methods.

*

```
Class Demo{
    private int salary;
    void setSalery(int s){ // setter method
        if(s<0){s=-s;}
        this.salary = s;
    }
    int getSalary(){        // Getter method.
        return this.salary;
    }

    PSVM (String args[]){
        Demo d = new Demo();
        d.Salary(-5000);
        int salary = d.getSalary();
        SOP (salary); → 5000.
    }
}
```

* Instance Variable :-

→ An instance variable is a variable which is declared in a class but outside of constructors, methods, or blocks.

→ Instance variables are created when an object is ~~int~~ instantiated (created). ~~and~~.

→ Instance variables are accessible to all the constructors, methods, or blocks in the current class.

→ Access modifier can be given to the instance variables.

Ex→ private name = "Hemant";

→ Instance variable have default values.

Ex→ int num; → default value 0

float per; → default value 0.0

String name; → defaut value NULL

Note :- at the time of object creation defaut constructor is called and this default constructor is responsible for providing the default values to instance variables.

→ Instance variables are not accessible in static members of current class.

→ Instance variable are created in the respect of object.

→
```
Class Student{
    int r-no=29; // instance variable
    Student(){
        r-no=30;
    }
    PSVM (String args[]){
        Student S1 = new Student();
        SOP (S1.name); → O/P- 30
    }
}
```

DTO → Data Transfer object (Bean class).
Pojo → ~~pa~~ plain old java object.

* Difference Between Pojo Class and Bean Class :-

Pojo → Every Pojo class is not a bean class but
Bean → Every Bean class considered as pojo class.
Pojo → Pojo class does not follow any special ~~ignored~~ restrictions other then those provided by Java Language Specification. but
Bean → Bean class followed special restrictions other then those provided by Java Language Specification (JLS).

~~Pojo~~ restrictions →
Pojo → data members can be public, private, protected and default (Pkg Private). but
Bean → Data members must be private.
Pojo → Member functions can be public, private, procted and default (Pkg private). but
Bean → member functions must be private.
Pojo → Pojo class can implement serializable interface but
Bean → it's mendatory to implement serializable interface in Bean class.

Pojo →

```
class Pojo{
    String name = "Hemant";
    Private int r-no = 1002;
    String getName(){
        return name;
    }
    int getRno(){
        return r-no;
    }
    PSVM (String args [])
    Pojo p = new Pojo();
    String name = p.getNamd();
    int r-no = p.getRno();
    sop(name); → Hemant
    sop(r-no); → 1002
}
}
```

Bean →

```
class Bean{
    Private String name = "Hemant";
    Private int r-no = 1002;
    Public String getName(){
        return name; }
    Public int getRollNo(){
        return r-no; }
    PSVM(String [] args){
        Bean b = new Bean();
        String name = getName();
        int r-no = getRollNO();
        sop(name); → Hemant
        sop(r-no); → 1002
}
}
```

this keyword :-

V1) this can be used to return the current class instance.

V2) this can also be passed as an argument inside constructor calling.

V3) this can be passed ~~an~~ as an argument inside function calling.

V4) this can be used to represent current class data members and members functions.

V5)

V6) this can be used to call current class constructor.

→ only from inside the constructor we can call another constructor of the current class.

→ this must be the first statement inside constructor.

* 'this' is a reference variable that refers to the current ~~class~~ object.

~~class~~ .

```
class Student {
    String name;
    int roll;
    Student (String name, int roll){
        this.name = name;
        this.roll = roll;
        this.show();
    }

    Student getInstance(){
        return this;
    }

    void Display (Student obj){
        sop(obj.name);
        sop(obj.roll);
    }

    void show (){
        sop("Inside show");
        Display(this);
    }

    Student (){
        this ("Hemant", 10);
        SOP("Inside default constructor");
    }

    psvm(String [] args){
        Student s = new Student();
        Student s2 = s.getInstance();
        sop(s1);
        sop(s2);
    }
}
```

(4)

(5)

(1)

(3)

(6)

O/P → Inside show
Hemant
10
Inside default constructor
Student @ 469966
Student @ 469966 } equal

```
class Parent {
        Parent (Child obj){
                SOP( obj.name);
                SOP (obj.age);
                SOP ("Inside parent constructor");
        }
}

class child extends parent {
        ~~Chi~~ name = "Andrew";
        Age = 19;
        child ( ) {
                super (this); (2)
                SOP("Inside child constructor");
        }
        PSVM(String [] args){
                Child c = new Child ();
        }
}
```

O/P -

```
Andrew
19
Inside parent constructor
Inside child constructor.
```

Parameter and Argument :-

```
Class Demo {
        void Add(int a , int b){
                Sop("a+b : " + (a+b));
        }
        PSVM(String args[]){
                Demo d = new Demo();
                d. Add (5, 6);
        }
}
```

Parameters or
Formal parameters or
Actual Arguments

Arguments or
Actual Parameters or
Formal ~~Parameters~~. Argument.

Constructor chaining:-

Constructor chaining refers to the ability to call a constructor from inside another constructor.

→ We can perform constructor chaining using-
(i) this keyword (same class)
(ii) super keyword (Different (Parent) class (Inheritence Imp.))

→ You can use a constructor chain either within the same class (using this keyword) or even with another one (using super keyword).

→ Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Function Calling

1>

Call by value :- (java , C, C++

→ in call by value we can pass actual values or variables or ~~or~~ constants.

→ whenever we pass the data, copy of original data is going to be passed. and all operations would be done on copied data.

→ changes done on copied data does not reflect on original data.

→
```
class Demo{
        void change(int a){
                a = 20;
                sop (a); → 20
        }
        int changeReturn (int a){
                a = 30;
                return a;
        }
        PSVM (String args[]){
                Demo d = new Demo();
                int a = 10;
                sop (a); → 10
                d.change (a);
                sop (a); → 10
                int c = d.changeReturn(a);
                sop(a); → 10
                sop (c); → 30
        }
}
```

copy created

copy created.

actual values are passing

changes are not reflecting on actual a

Constructor :-

→ Constructor is a special member type function.

→ the name of constructor must be same as class name.

→ Constructor does not return any value not even void.

→ the initial work of constructor is to construct object.

→ Constructor is responsible for providing default initial values to data members.

→ Constructor chaining is also possible in java using this keyword. it means we can call another constructor from a constructor of present (current) class without creating any object.

→ Types of Constructor -

1) Default constructor (No argument constructor or Non-Parameterised constructor)

2) Parameterised constructor (

3) Copy Constructor

i) Shallow copy

ii) Deep copy.

→ When we create a parameterised constructor and we want to call default constructor in that case it's mendetory to create a default (user defined) constructor.

→ Every time an object is created using the 'new' keyword, atleast one constructor is called.

→ A constructor in java can not be abstract, final, static or synchronized.

→ Access modifiers can be used in constructor declaration to control its access.

→ A constructor can be overloded but can't be overridden.

→ there are no "return value" statement in the constructor, but the constructor return the current class instance. we can write 'return' inside the constructor.

*. Types of Constructor :-

(i) Default Constructo (No-Argument constructor)
(ii) Parameterized constructor
(iii) Copy constructor.

(i) Default Constructor :-

→ Default constructor does not take any arguments.

→ A default constructor in java is created by the compiler itself when a programmer doesn't create any constructor.

→ the purposase of the default constructor is to initialize the attributes of the object with their default values.

→ programmer can also create these own constructor.

→ If I created only a parameterized constructor but I also want to use default constructor then in that case we must have to create an default constructor.

→ because compiler says "If you can create your parameterized constructor then you can also create your default constructor."

(ii) Parameterized Constructor :-

→ A constructor that has parameters (that takes arguments) is known as parameterized constructor.

→ Parameterized constructors can be more then one with different parameters.

→ we can call ~~a~~ default constructor from parameterized constructor using 'this' keyword.

(iii). Copy Constructor :-

→ Copy constructor is a constructor is a constructor that creates ~~and~~ an object using another object of the same java class.

→ Java doesn't create a default copy constructor, if you don't writ your own.

→ Why copy constructor is required?

- Sometimes, we face a problem where we required to create a extra copy of an existing object of the class.
- There is also a condition, if we have made any changes in the copy it should not reflect in the original one and vice-versa.

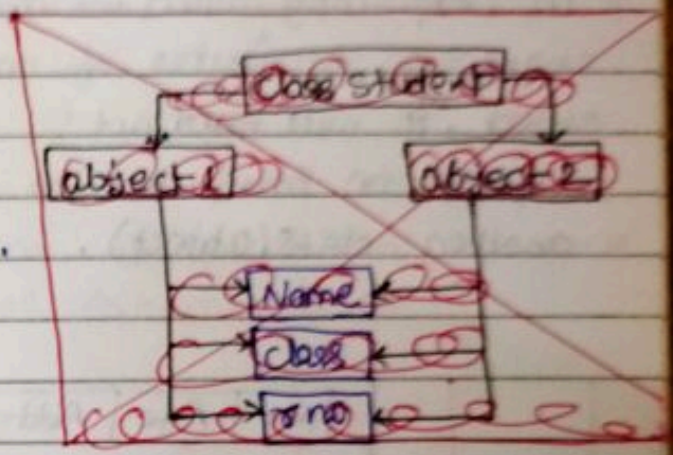
  for such cases java provides concept of ~~cocep~~ copy constructor.

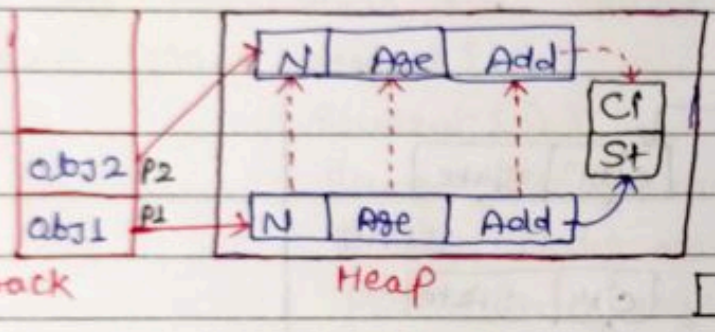
→ Copy constructor returns a duplicate copy of an existing object of the class.

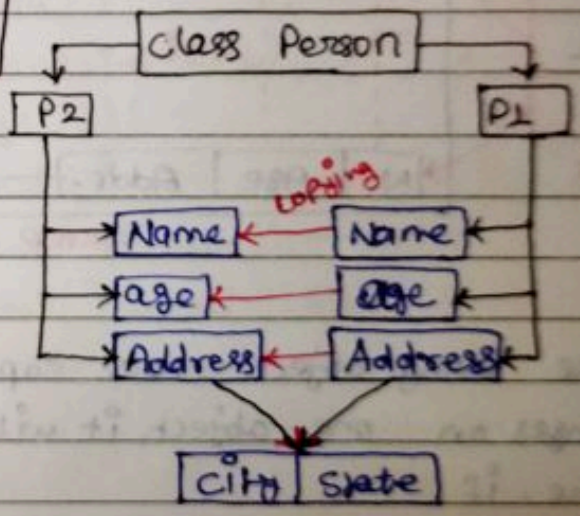
→ Basically two types of copy constructors are ~~used~~ Avilable in java:-
(i) Shallow Copy Constructor
(ii) Deep Copy constructor.

(i) Shallow Copy Constructor :-

→ In shallow copy Both objects are share same memory area.







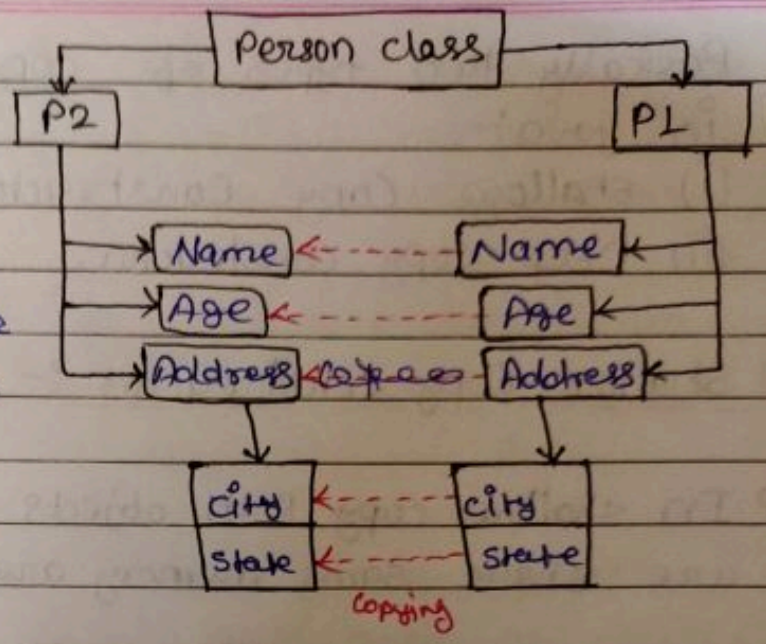
→ here Address is shared between both P1 and P2 that's why if we make changes on address of P1 then it will also reflect on Address of P2.

* here Address is Reference variable.

→ In shallow copy when we make change on Attributes of one object, It will also reflect on Attributes of another object.

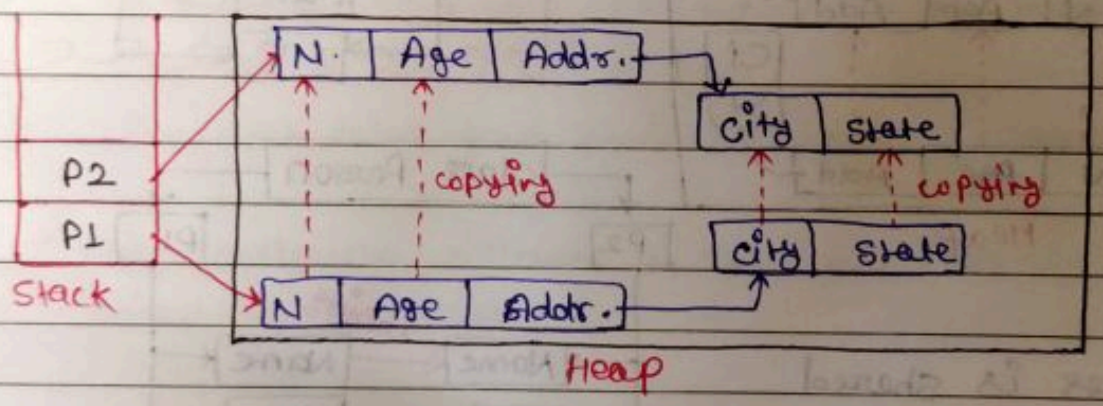
→ here, in example if we make changes on Name and Age then it will not reflect on both because ~~they are~~ both ~~sharing~~ ~~same~~ memory area.
have different.

(ii) Deep Copy :-



→ In Deep Copy Every object have saperate memory.

→ In Deep Copy when we make changes on Attributes of one object, it will ~~need~~ not reflect on attributes of another ~~class~~ (object).



→ Here every object have saperate ~~make~~ memory thats why if i ~~ot~~ make changes on one object, it will not reflect on another object.

→ here, if

#. Constructor Overloading :-

→ in java, we can make n number of constructors according to our requirement.

→ Every constructor have must be different parameters.

→ if no. of Parameters are same then ~~it must have~~ sequence of Parameters must be different or datatype of Parameters must be different.

→ Constructor overloading does not depends on parameter name but it depends on no. of parameter, sequence of parameters and datatype of Parameters.

Ex→

```
class Student {
    Student ( ) {      };
    Student (String name) {      };
    Student (String name, int r_no) { };
    PSVM (String args[]) {
        new Student ( );
        new Student ("Hemant");
        new Student ("Hemant", 43);
    }
}
```

→ when we have multiple constructor in a class with different parameters then compiler will call automatically that constructor who satisfies the Arguments passing at the time of object creation.

this is known as Constructor overloading.

Array of Objects:-

→ Array of object is used to store multiple objects of a single class.

→ Declaration of array of object:

Syntax →

<C-name> <arr-name>[] = new <C-name>[<no. of objects>];

Product P[] = new Product[5];

{ P is also called as Refference array because at the current time there is not any object is P }

Eg:

0	1	2	3	4	← indexing
null	null	null	null	null	← elements.
P[0]	P[1]	P[2]	P[3]	P[4]	

→ Initialization of array of object:

```
for (int i=0 ; i< P.length ; i++){
        <arr-name>[i] = new <class-name>( );
}
```

0	1	2	3	4	← indexing
<class-name>@5de44f73	...	..	..	---	← objects.
<arr-name> [0]	[1]	[2]	[3]	[4]	

now the array is known as array of objects.

#. Variable Arguments (Varargs) :

```
Class Demo {
        Static void add(int ... a) {
                int sum = 0;          → Varang
                for(int a1 : a) {
                        sum += a1;
                }
                SOP(sum);
        }
        PSVM (String args []) {
                add(100); → 100
                add(1, 2, 3, 4); → 10
                add ( ); → 0
        }
}
```

→ Varargs is a Method (technique) to pass n number of similar type of arguments and hold all the arguments in a single variable (Parameter);

→ Varargs applied on same datatype of arguments.

→ Varang must be the last parameter in function definition.

→ We can not create more than one varang as parameter.

→ When we pass values then varang internaly creates an array and it holds all values in an array.

→ The varargs allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem.

→ If we don't know how many arguments we will have to pass in the method, varargs is the ~~method~~ better approach.

Syntax →

<return_type><method_name>(datatype ... variable name) { }

#. final keyword :-

→ The final keyword in java is used to make constant.

→ the final keyword in java is used to restrict the user.

→ if you make any array (reference variable) as a final then in that case that array (reference variable) can not reffer to any other Refference (object).

→ if we use final keyword with

a) Variable : we can't update value of final variable

b) Method : we can't override final method.

c) class : we can't extends final class.

→ we can inherit final methods but we can't override it.

→ a final variable that is not initialized at the time of declaration is known as blank final variable.

→ we can initialize the blank final variable but only inside the constructor.

→ a static final variable that is not initialized at the time of declaration is known as static final blank variable. It can be initialized only in static block.

→ If you ~~det~~ declare any parameter as final, then it is known as final parameter and we can't change the value of it.

→ we can't declare a constructor final, because constructor is never inherited.

```
class Demo{
    final int num;
    Demo(){
        num=10;
    }
    final void run(){sop("running....");}
    Static final int num2;
    Static{  num2 = 20;
        sop("num2:"+20);
    }
}

class Main extends Demo{
    PSVM (———){
        SOP ( Demo. num2);      → 20
        Demo d = new Demo();  →
        SOP (d. num);           → 10
        d. run();               → running....
    }
}
```

Static keyword :-

→ The static keyword in java is used for memory management mainly.

→ We can apply static keyword with variables, methods, blocks, and classes or nested classes.

* Java static variables :-

→ If you declare any variable as static, it is known as static variable.

→ The static variables can be used to refer the common property of all objects (which is not unique for each object).

→ Static variable gets memory only once in the class area at the time of class loading. (It makes your program memory efficient).

→

→ ~~Class~~ static variables can't access not static variables.

* Java static Methods :-

→ If you apply static keyword with any method, it is known as static method.

→ A static method belongs to the class rather than the object of class.

→ A static method can be invoked without the need for creating an instance of a class. (It means we can call the static methods ~~uni~~ using class name).

→ A static method can access the static data member and can change the value of it.

→ the static method can not use non static data member or call non-static method directly.

→ this and super can not be used in static context. (they can not refer to this or super in any way).

DATE
PAGE

* Why is java main method static?
→ It is because the object is not required to call static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory

→ inside ~~statide~~ static method we can access only static variables. or members.

* Static block :-
→ It is used to initialize the static data members.
→ It is executed only one time before the main method at the time of class loading.
→ the execution sequence of static block is top to bottom.
→ we can't access non-static members inside static ~~bock~~ blocks.

*. Can we execute a program without main() method?
→ No, we can't execute any program without main() method, but It was possible till JDK 1.6, since JDK 1.7 it is not possible to execute a class without the main method.

* Static class :-
→ A class can be made static only if it is a nested class.
→ (root) ~~one~~ outer class can't be declared as static.

* Some General Points about static keyword:-

→ we can't use static keyword inside methods.

→ If we create multiple objects then static things are executes only one time at the time of class loading.

→ we can access to static members without creating any object only inside this class but outside this class we can access access static members with the help of class name.

→ we can access only static members inside static methods, blocks or variables in a class.

→ Static blocks and static variables are executed in order they are present in a program.

→ static members are created in the respect of class.

→ static methods can be overloaded but they can't be overridden.

→ we can not create static member and instance members with same name in a single class.

* Static keyword is used in the respect of class members.

→ root class can't be static.

```
class Demo {
    static int num;
    static {
        num = 10;
    }
    static void run(){
        sop("running .....");
    }
    static class Nested{
        static void show(){
            sop("showing .....");
        }
    }
    psvm (String args[]){
        sop("num : " + num); → 10
        run();               → running ---
        Nested.show();       → showing ---
    }
}
```

Initializer Block :- (Instance Initializer block)

→ Instance Initializer block is used to initialize the instance data member.

→ Initializer block runs each time when object of the class created.

→ we can perform any operation other then initialization, inside of Initializer block.

→ Initializer block must be inside of a class but outside of methods and constructors.

→ The contents of initializer block are executed (before ~~comes the~~ execution of constructor's contentent) whenever any constructor is invoked.

→ it means when constructor invoked then the control of program goes on Initializer block and execute the code of initializer block then control back on constructor and executes the code of constructor.

→ the order of execution of Initializer blocks is top to down.

→ if child class extends parent class then if we create an object of child class then first "child class constructor invoked → Parent class constructor invoked → parent class constructor code executed → child class initializer block executed → child class constructor code executed.

```
class Student {
    String name ; int r-no = 10;
    {
        name = "Hemant";
        r-no = 10;
        SOP("Initialiser block called");
    }
    Student () {
        SOP("Inside constructor");
        SOP (name);
        SOP (r-no);
    }
    PSVM (String [] args){
        Student S = new Student();
        SOP("Inside main");
    }
}
```

→ initializer block.

Output:

```
Initialiser block called
Inside constructor
Hement
10
Inside main.
```

```
class Parent {
    Parent( ){
        SOP("Inside Parent class constructor");
    }
    {
        SOP("Inside Parent Initializer block");
    }
}
class child extends Parent {
    Child( ) { //super();
        SOP("Inside child class constructor");
    }
    {
        SOP("Inside child class initializer block");
    }
    PSVM(String args[]){
        Child c = new child();
        SOP ("Inside main method");
    }
}
```

Output:

```
Inside Parent Initializer block
Inside Parent class constructor
Inside child class initializer block
Inside child class constructor.
Inside main method.
```

Garbage Collector (Collection):-

Garbage collection is the process ~~of~~ by which java programs perform automatically memory management.

* need of Garbage collection:-
In C/C++ Programers are responsible to creation and distruction of objects. In C/C++ programming programes are responsible to allocate and deallocate the memory. some time developers are forget to deallocate the memory that why program had thrown outoffmemory error.
Java programming comes in the market with the solution of this problem that is Garbage collection.

→ in java, Garbage means unreferenced objects.

*→ to reclaim the unused memory, in C we use free(); function, in C++ we use delete(); function. but in java it is perforformed automatically but we can also request to the Garbage collector to reclaim the unused memory by giving instruction System.gc();

→ JVM is ~~rs~~ responsible for calling your garbage collector which resides inside the execution engine.

→ Garbage collector of JVM, collects only those unreferenced objects ~~as~~ that are created by new keyword.

DATE
PAGE

* Types of Activities in Java Garbage Collection :-

(i) Minor or Incremental Garbage Collection :-

It is said to have ~~occurred~~ occurred when unreachable objects in young generation, heap memory are removed

(ii) Major or Full Garbage Collection :-

It is said to have occurred when unreachable objects in the young generation, are copied into the old generation or permanent generation, heap memory are removed. old generation garbage collection happens less frequently compare to the young generation garbage collection.

* Ways to make an object eligible for Garbage collector :-

→ Nullifying the reference variable
→ Re-assign the ~~reffer~~ reference variable
→ An object created inside the method.
→ Creating the anonymous object.
→ Island of isolation.

* Ways to requesting JVM to running Garbage collector :-

(i) Using System.gc() method :- System class contains static method gc() for requesting JVM to run Garbage collector.

(ii) Using Runtime.getRuntime().gc() method :-

Runtime class allows the application to interface with the JVM in which the application is running.
and by ~~by~~ using gc() method we are requesting JVM to call garbage collector.

* Finalization:-

Just before destroying an object, Garbage collector calls finalize() method on the object to perform cleaning activities. once finalize() method completes Garbage collector destroys that object.

→ finalize() method is present in object class with following prototype:

protected void finalize()

we can override ~~fini~~ finalize() method for ~~pet~~ performing our cleanup activities like closing connection from the database.

→ the finalize() method is called by Garbage Collector, not JVM. However, Garbage collector is one of the modules of JVM.

→ the finalize() method is never invoked more than once for any object.

→ object class finalise() method has an empty implementation. Thus, it is recommended to override the finalise() method to dispose of system resources or perform other cleanups.

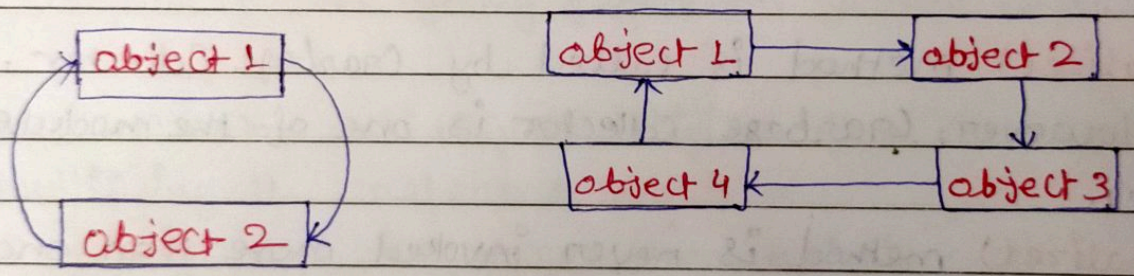
→ If an uncaught exception is thrown by the finalise() method, the exception is ignored and the finalization of that object terminates.

* Island of Isolation in java :-

→ When object 1 references object 2 and object 2 references object 1. Neither object 1 nor object 2 is referenced by any other object. that's an island of isolation.

→ an island is a group of objects that reference each other but are not referenced by any active object in the application.

→ a single unreferenced object is an island of isolation.



→ if any one connection destroyed then all connections will be destroyed and all objects will be dereferenced and that's why all objects will be eligible for garbage collection.

```
class Demo {
    public void finalize(){
        SOP("Object is Garbage Collected");
    }
    PSVM(String args[]){
        Demo d1 = new Demo();
        Demo d2 = new Demo();
        System.gc();
    }
}
```

o/p: object is Garbage collected
Object is Garbage Collected.

Design Patterns:-

→ Design patterns are well proved solution for solving the specific problem/task.

→ We must use the design patterns during the analysis and requirement phase of SDLC (S/w dev. Life cycle).

→ design patterns are programming language independent strategies for solving the common object-oriented design problems.

→ Categorization of design patterns:

Design Pattern

- (1) Core java (or JSE) Design Pattern
  - (1) Creational Design Pattern
  - (2) Structural Design Pattern
  - (3) Behavioral Design pattern
- (2) JEE Design pattern
  - (1) Presentation Layer Design Pattern
  - (2) Business Layer Design Pattern
  - (3) Integration Layer Design pattern.

(1) Core Java Design Patterns :- (JSE)

Full form of JSE is " Java

→ this are divided into three further parts:-

(1) Creational Design Pattern:-

→

→ Creational Design Patterns are concerned with the way of creating object.

→ this design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

→ types of creational Design patterns—

(i) Factory Method Pattern
(ii) Abstract factory pattern
(iii) Singleton pattern ←
(iv) Prototype Pattern
(v) Builder pattern
(vi) object pool pattern.

(2) Structural Design Pattern:-

→

→ Structural Design Patterns are concerned with how classes and objects can be composed, to form larger structure.

→ The structural Design patterns simplifies the structor by identifying the relationship.

→ types of structural Design pattern:-

(i) Adapter pattern
(ii) Bridge pattern
(iii) composite pattern
(iv) Decorator Pattern
(v) Facade Pattern
(vi) flyweight Pattern (vii) Proxy Pattern

(3) Behavioral Design Pattern:-

→ Behavioral design patterns are concerned with the interaction and responsibility of objects.

→ In these design patterns, the interaction between the objects should be in such way that they can easily talk to each other and still should be loosely coupled.

→ types of behavioral design pattern:-

1) Chain of Responsibility Pattern
2) Command pattern
3) Interpreter Pattern
4) Iterator pattern
5) memento pattern
6) Madiator pattern
7) observer pattern
8) State pattern
9) Strategy pattern
10) Templete pattern
11) Visitor pattern
12) Null Object.

Singleton Class :-

Singleton class is a class which can have only one object (instance) at a time and after first time if you try to create another object then in that case new reference variable also points to the previosly created object.

→ Rules of create singleton class:

rule 1) make all constructors private.

rule 2) make a static method whose return type is the object of this singleton class.

→ Purpose of Singleto class:

→ the primary purpose of java singleton class is to restrict the limit of the number of object creations to only one.

→ Stop memory wastage.

→

rule(3)

→ Ensure that the instance of singleton class is stored as a private static variable. (~~[illegible]~~).

→ there are various ways ~~to~~ with the help of which we can create a single object of a ~~cla~~ singleton class, we follow some of the ways given bellow —

(i) Early Initialization

(ii) Lazy Initialization.

(i) Early Initialization :-

→ creation of object at ~~class~~ the time of class loading.

```
class Singleton{
    static private Singleton instance = new Singleton();
    private Singleton () { };
    static Singleton getInstance (){
        return instance;
    }
}
```

```
// Egar Initialization through static block.
class Singleton1{
    static private Singleton instance;
    private Singleton1() { };
    static {
        instance = new instance();
    }
    static Singleton getInstance (){
        return instance;
    }
}
// main
class Main {
    PSVM ( String [] args ) {
        Singleton obj = Singleton.getInstance ();
        Singleton1 obj1 = Singleton1.getInstance ();
    }
}
```

(ii) Lazy Initialization :-

→ Creation of instance when required.

```
class Singleton{
        static private Singleton instance;
        private Singleton (){};
        static Singleton getInstance(){
                if(instance == null){
                        instance = new Singleton();
                }
                return instance;
        }
}
//main
class Main{
        psvm(String [] args){
                Singleton instance = Singleton.getInstance();
        }
}
```

NESTED CLASS :-

In java it is possible to have a class inside a class and such class is known as nested class.

→ Nested class in java allows you to logically group multiple classes at one place to use ~~the~~ ~~achieve~~ (achieve) the concept of Encapsulation. and to create more maintainable and readable code.

```
class OuterClass        → Enclosing class (outer class)
{
    class NestedClass   → Nested class (inner class)
    {
    ==========          → member class
    }
}
```

→ Nested class is also considered as a member of Enclosing class.

→ The scope of Enclosing class lies under Enclosing class.

→ as Nested class is a member of Enclosing class so, it can be public, private, protected, default (Pkg private), final, abstract or static.

→ Enclosing class can be default and public. but it can't be static.

→ Nested class can access the member of Enclosing class (Private members also can be accessible).

→ Enclosing class can not access the members of Nested class.

Nested Class

① Static Nested class

② Non-static nested class (inner class)

(i) Anonymous class. (ii) Local class. (iii) member class.

(1) Static Nested class:-

→ for making any class as a static Nested class we need to write static keyword followed by the class name.

→ A static class that is created inside a class (and outside of methods) is called a static nested class.

→ Static Nested class can not access non-static data members and methods of Enclosing class.

→ It can be accessed by outer class name.

→ Static nested class can access static data member of the outer class.

→ the creation of object of static nested class does not depends upon the object of Enclosing class.

→ Creating object of static nested class syntax:

```
OuterClass.StaticNestedClass nestedObject = new
        OuterClass.StaticNestedClass();
```

Ex->

```
class Demo{
    Static class Inner{
        Static void show(){
            SOP("showing ....");
        }
        Void run(){
            sop("running ....");
        }
    }
    PSVM(String args[]){
        Inner.show();      -> showing....
        Inner i = new Inner();
        i.run();  -> running....
    }
}
```

(2) Non-Static Nested Class :- (Inner class) :-

→ the creation of object of inner class or non-static ~~class~~ nested class depends upon the object of Enclosing class (outer class).

→ to instatiate an inner class, you must first instatiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = OuterObject.new InnerClass();
```

types

(2.1) Member class :-

→ A non-static class that is created inside a class but outside a method is called member inner class.

→ it is also known as regular inner class.

→ It can be declared with access modifiers like public, default, private and protected.

→ We can't create static data members (without final keyword) inside (member inner class) or non-static nested class.

→ modifier 'static' is only allowed in constant (final) variable declaration.

→ we can't create static method inside member class because ~~it depends on~~ static method will depends on object of outer class.

(2.2) Local Class :-

→ A class created inside a method is called local Inner Class.

→ Local inner classes can be defined inside a loop block, or if block or else block or initializer block.

→ Local Inner classes are not a member of any enclosing class.

→ Local Inner classes belongs to the block they are defined within that's why any access modifier can't (not) associated (used) with them. However, they can be marked as final or abstract.

→ If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

→ Local inner class can not be invoked from outside the method.

→ Local inner class cannot access non-final local variable till JDK 1.7 since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

→ A local class has access to the members of its enclosing class.

→ Local inner class can extends an abstract class or implement an interface.

→ Local inner class can not be static.

→ Inside Local class we can't create static method.

→ Inside Local class we can create final static variable.

```
class Demo {
    class Inner {
        void show() {
            SOP("show");
        }
    }
    PSVM(String [] args){
        class Local {
            void run() {
                SOP("run");
            }
        }
        Local l = new Local();
        l.run(); → run
        Demo d = new Demo();
        Demo.Inner i = d.new Inner();
        i.show(); → show.
    }
}
```

(2.3) Anonymous Inner Class:-

Java anonymous inner class is an inner class without a name and for which only a single object is created.

→ An anonymous inner class can be useful when making an object with certain "extras Functionalities" such as method ~~overloading or~~ method overriding of a class or interface, without making a subclass of this class.

→ a class that has no name is known as anonymous inner class in java.

→ Anonymous inner class can implement only one interface at a time.

→ Anonymous inner class can extend a class or can implement an interface but not both at a time.

→ we can't create any constructor of anonymous inner class because anonymous class does not have any name but write → ~~default constructor called implicitly. (it means default constructor is present in Anonymous Inner class)~~

→ An anonymous class has access to the member of its enclosing class.

→ An anonymous class can not access local variables in its Enclosing scope that are not declared as final or effectivelly final

→ we can declare the following in anonymous classes as follows:

(i) Fields

(ii) Extra methods (even we can't implement any method of the supertype)

~~(iii) Instance Initializers (initializer block)~~

~~(iv) local classes~~

(v) Static block (static initializers) or

(vi) member interface.

→ an anonymous class can have static members but they must have constant.

→ In anonymous class Inheritence is compulsory. without Inheritence Anonymous ~~co~~ class can't created.

→ we can't add new functionality in anonymous class but we can modify existing functionality (it means overloading is not possible but overriding is possible).

→ we can create a new class ~~fo~~ or local class in Anonymous class but we can not create new ~~variables~~ ~~cod~~ methods, and we can also create new instance variables.

→ we can create initializer block in Anonymous class.

→ we can create new ~~vari~~ instance variable inside anonymous class but we can't access this variable from outside of this anonymous class.

* there are three ways to create Anonymous class-
  1) by Interface
  2) by Abstract class
  3) by Normal class.

(1) By Interface :

```
Interface Etable{
        void eat();
}
class Demo {
        psvm(String args[]){
            Etable e = new Etable(){
                        public void eat(){
                                sop("nice fruit");
                        }
            };
            e.eat(); → nice fruit.
        }
}
```

(2) By Abstract class:

```
abstract class Food {
        abstract void eat();
}
class Test {
        public static void main (String args[]){
                Food f = new Food (){
                        public void eat (){
                                sop(" nice food");
                        }
                };
                f.eat(); → nice food.
        }
}
```

(3) By Normal Class:-

```
class Food {
        public void eat (){
                SOP(" nice food");
        }
}
class Main {
        psvm (String args[]){
                Food f = new Food(){
                        @Override
                        Public void eat(){
                                SOP(" Bad food");
                        }
                };
                f.eat(); → Bad food.
        }
}
```

#. Modifiers :-

Modifiers are keywords that are used to specify the access level, scope and behaviour of classes methods ~~or~~ and variables.

→ Modifiers are two types in java :-
(i) Access modifiers :- Page no.-
(ii) Non- access modifiers :-

* Non-Access Modifiers :-

Non- ~~Stat~~ Access modifiers are used to specify additional behaviors and characteristics of classes, methods and variables.

→ There are servel types of Non- Access modifiers in java including:
Static, final, abstract, synchronized, volatile, Transient etc.

'Extends' keyword :-

the 'extends' keyword is used to create a subclass that inherits properties and methods from a superclass.

→ The 'extends' keyword is used in a class declaration to indicate that the class is a subclass of another class.

```
class Animal {
  }
  class Dog extends Animal {

  }
```

→ A subclass can override the methods of its superclass, and can also call methods of its super class using 'super' keyword.

'Super' keyword :-

→ In first line of child class constructor super(); keyword is always written implicitly.

→ super(); must be the first statement inside child class constructor for invocking parent class constructor.

→ we can call parent class constructor using super() keyword only from inside of child class constructor.

→ super can be used to refer imidiate parent class instance variables.

→ super super can be used to invoke imidiate parent class methods.

→ super can be used to invoke immediate parent class constructor

→ super keyword in java is a reference variable that is used to refer parent class constructor.

```
class Parent {
        String name = "Andrew";
        void show(){
                SOP("Inside show in parent");
        }
        Parent(){
                SOP("Inside constructor ~~child~~ of Parent");
        }
}
class Child extends Parent {
        String name = "Mathew";
        void display(){
                SOP("Inside Display of child"); SOP(super.name);
                super.show();
        }
        Child(){ //super();
                SOP("Inside constructor of child");
        }
        PSVM (String [] args){
                child c = new child();
                c.display();
                c.show();
                SOP(c.name);
                Parent p = new Parent();
                p.show();
                // p.display(); // -> throws error
                SOP(p.name);
        }
}
```

O/P -> Inside constructor of ~~child~~ parent
Inside constructor of child
inside display of child
Andrew
Inside show in Parent
~~Inside show in Parent~~
mathew
Inside constructor of parent
Inside show in Parent
Andrew.

Inheritence :-

Inheritence in java is a mechanism in which one object aquires all the properties and behaviors of a parent object.

→ Inheritence Represents the IS-A Relationship which is also known as a parent-child relationship.

→ the main Advantage of Inheritence is code Reuseability.

→ by using 'extends' keyword we can implement Inheritence.

→ Method overriding is achieveable only through Inheritence.

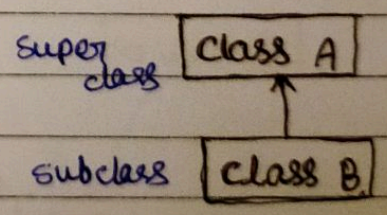
→ by using Inheritence we can also achieve the abstraction

→ In java programming, Inheritence are basically 1 types

(i) Single Inheritence
(ii) Multi level Inheritence
(iii) Hierarchical Inheritence
(iv) Hybrid Inheritence
(v) Multiple Inheritence.

(1) Single Inheritence :-

when a class (subclass) Inherit the properties of another class (parent class), it is known as single Inheritence.

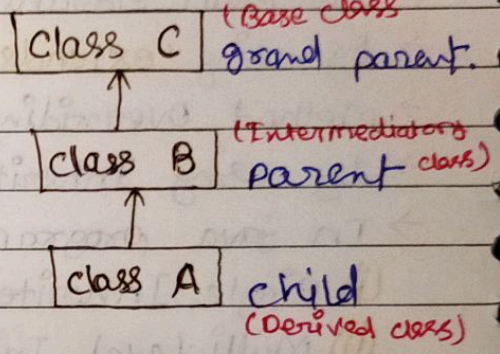


* the Accessibility of overrided methods in child class must be equal or more Accessible in child class compare to parent class.
* Inheritence creates a tight coupling between the super class and subclass.

(2). Multilevel Inheritence :-

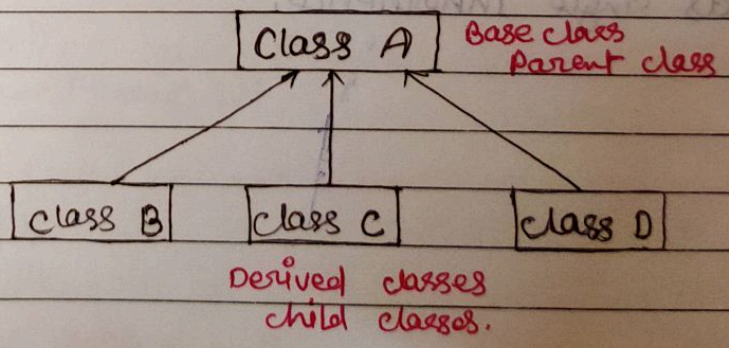
when a child class (A) inherites the properties of parent class (B) and parent class inherits the propertis of any grand parent class (C), it means when there is a chain of inheritence, it is known as multilevel Inheritence.

→ here class A containing the property of both upper classes ~~A~~ B and C.



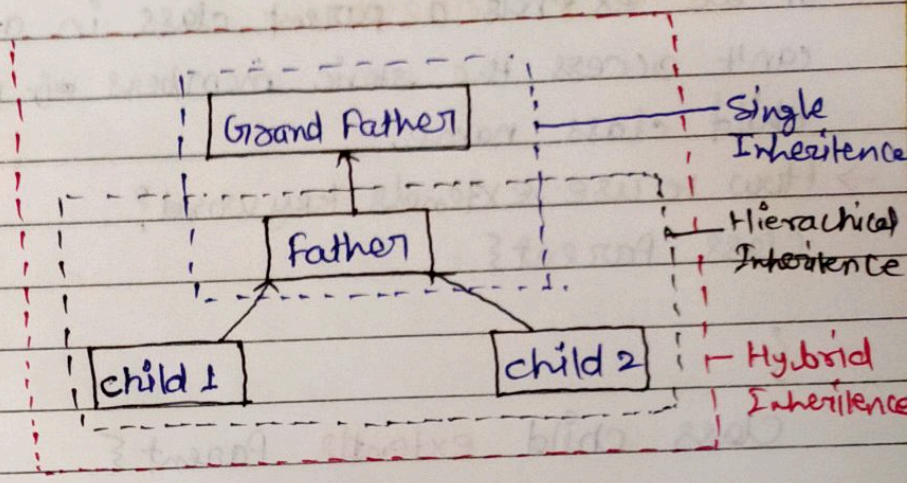
(3). Hierarchical Inheritence :-

when a child class (~~B~~ A) inherits the proterties of a parent class (~~B~~ A) and anothother child class (C) also inherits the properties of the Same parent class (A), it means when two or more classes inherits a single class, it is known as Hierarchical Inheritence.



(4.) Hybrid Inheritence :-

→ When two or more types of ~~the~~ Inheritence are combined, then it is called Hybrid Inheritence.

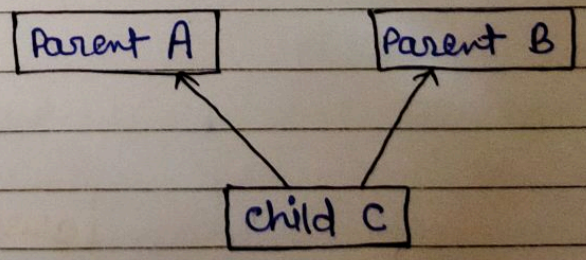


(5). Multiple Inheritence :-

* "Mutiple Inheritence is not allowed in java but we can achieve ~~at~~ the functionality of multiple Inheritence using Interfaces."

"because we can ~~exp~~ implement multiple Interfaces in a single class at a time."

→ When a child class inherits the properties of two or more parents then it is called Multiple Inheritence.



→ if we extends a parent class in a child class then we can't access the static members of parent class with (using) child class name.

→ How to use extends keyword?

```
Class Parent{

}

Class child extends Parent{

        psvm(String[] args){

        }
}
```

Abstraction :-

Abstraction is the process of hiding implementation details and showing only functionalities to the user.

→ It works on the concept of data hiding

→ we can achieve abstraction with the help of Interface and Abstract classes.

→ If we want to achieve abstraction on a single class then we go with the concept of Encapsulation.

→ when we want to achieve ~~con~~ abstraction ~~of~~ on multiple classes then in that case we go with the concept of abstract classes.

→ there are two ways to achieve abstraction:

1. Abstract Class (0 to 100%)
2. Interface (100%)

(1) Abstract Class :-

A class which is declared as abstract is known as an abstract class.

→ for making any class as an abstract class. we need to write abstract keyword. followed by the class name.

→ it's not mandatory for the abstract class to have an abstract method inside it.

→ an abstract class can contains abstract methods and concrete methods.

→ an abstract class can also contain constructor and ~~the~~ this can be called only from their concrete subclass.

→ abstract class can't be instantiated ~~because~~ which means we can't create object of abstract class because it contains abstract methods.

* Abstract Method : whose definition part is absent.
* Concrete Method : whose definition part is present with declaration of this method.

→ it is mandatory for the child class who extends abstract parent class, to give the implementation (definition part) of Abstract methods of ~~child~~ parent class.

→ if child class not giving the implementation of abstract methods of parent abstract class, then child class must be abstract.

→ abstract class provides Reference to their child class where child class can create their own object (upcasting)

```
Parent obj = new child();
```

in reference with abstract parent class.

→ Why we use abstract class?

- for achieving some functionality forcefully in child class then in that case we are going to use abstract class.
- For achieving the abstraction.

→ abstract methods can be overloaded.

→ abstract class can have final methods.

→ abstract class can have static method also.

```
abstract class Bike {
        Bike(){
                SOP("Bike is created");
        }
        abstract void run();
        void changeGear(){
                SOP("Gear Changed");
        }
}
class Honda extends Bike{
        void run(){
                SOP("Running---");
        }
}
class Test {
        public static void main(String args[]){
                Bike obj = new Honda(); → Bike is created
                obj.run(); → Running---
                obj.changeGear(); → Gear changed.
        }
}
```

Reference Typecasting :-

→ reference typecasting are two types - (i) Upcasting
(ii) Downcasting

(i) upcasting :-

→ when we create Parent type reference variable and in this variable we store the ~~of~~ Reference of child class object then this process is known as upcasting.

```
Parent obj = new Child();
```

→ Whe we want to call the methods using this obj then first compiler checks the availability on compiletime of that method if method is not present in parent class compiler ~~throws~~ gives error but if method is present then successfully compiled. then at run time if child class have that method then child class methods runs if child class have not ~~any~~ that method then parent class method will be run.

(ii) Downcasing :-

→ we can downcast only upcasted variables, it means without upcasting downcasting is not possible.

→ Parent obj = new Child(); (upcasted)
Child obj = new Child(); (After downcasting)

→ in downcasting we provides same class reference variable that class object is stored.

#. Dependency Injection :-

The Dependency Injection is a design pattern that supports loose coupling.

→ Dependencies are "injected" into a class from external sources rather than being created within the class itself.

→ Dependency Injection used to implement Inversion of Control (IoC) in Software System.

* Benifits of Dependency Injection :-

① Loose Coupling
② Modularity
③ Testability
④ Reusability
⑤ Single Responsibility.

* Types (Approaches) to Dependency Injection :-

(1) Constructor Injection :-

→ Depencies are proved through a class constructor.

→ Dependencies are passed as a parameter to the constructor when instance is created.

→ Constructor injection automatically inject the dependency when instance is created.

(2) Method Injection or Setter Injection:-

→ Dependencies are proved through a setter method, or other method that accept dependency as a parameter.

→ setter Injection inject dependency when we call it.

```
class Demo {
    Address address1;
    Address address2;
    Demo(Address a){          ← Constructor Injection
        address1 = a;
    }
                                  ↗ dependency.
    public void setDependency(Address a){   ← setter injection
        address2 = a;
    }
    PSVM (String args[]){
        Address a1 = new Address("Indore");
        Address a2 = new Address("Bhopal");
        Demo d = new Demo(a1);
        d.setDependency(a2);
    }
}
```

* Difference Between Constructor and Setter Injection:-

(1) Constructor automatically inject dependencies when we create an instance even if it is not required, but Setter injection inject dependency when we call it.

#. instanceof Operator :-

→ instanceof operator simply represents weather the reference variable is containing the given type of object reference or not.

```
Demo obj = new Demo();
```

object

same or not

Ex→ if(obj instanceof Demo){ return true; };
else{ return false; };

→ it always returns true or false.

Coupling in Java :-

→ Coupling is Nothing but the dependency of one class on the other class.

→ Coupling refers to the usage of an object by another object.

→ there are two types of coupling in java-

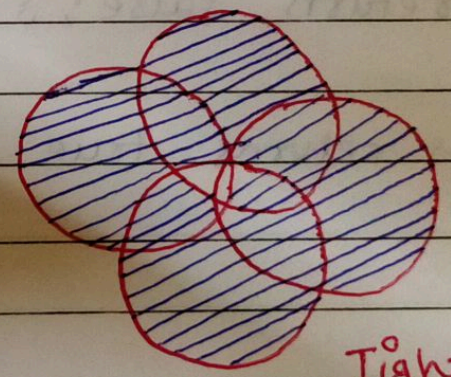
1.) Tight Coupling

2.) Loose Coupling.

(1.) Tight Coupling :-

→ Occurs when classes have strong dependecies on each other.

→

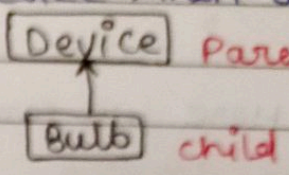


Tight coupling.

Relationship in Java :-

→ a relationship in java means different relations between two or more classes.

Ex→ if a class Bulb inherits another class Device, then we can say that Bulb is having IS-A Relationship with Device.
which means Bulb is a Device.



→ in java, we have two types of relationship:-
1.) Is-A Relationship
2.) Has-A Relationship.

(1) Is-A Relationship :-

→ Whenever one class inherits another class, it is called Is-A Relationship.
→ Is-A Relationship is wholly related to Inheritance.
→ Is-A Relationship can simply be achieved by using extends keyword.
→ Is-A ~~Relati~~ Relationship is additionally used for Code reusability in java and to avoid code redundency.
→ Is-A relationship is unidirectional, which means we can say that the "Bulb is a Device" but vice versa; a "device is a bulb is not possible" since all the devices are not bulbs.
→ Is-a relationship is tightly coupled, which means changing on one entity will affect on another entity.
→ Is-A relationship can simply be achieved by extending an interface or class by using extend keyword.

→ Ex→ Class Device {

}

Class Bulb extends Device {

}

→ Here we can say "Bulb is-A Device".

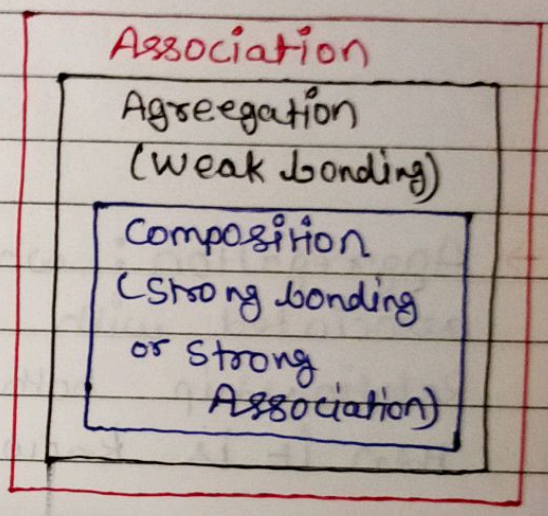
(2) Has-A Relationship :-

When one class Associated with another class, it is known as Has-A Relationship.

* Association : Association is the relation between two separate classes which establishes through their object.

→ Composition and Aggregation are two forms of Association.

→ a relationship where an object has a reference to another instance of same class or another class, is called Has-A relationship.



→ Ex-

if a object car has reference to object Engine then we can say "Car Has-A Engine".

```
Class Engine {
}
```

```
class Car {
    Engine e = new Engine();
}
```

Here "Car Has-A Engine."

→ Has-A Relationship are loosly coupled.

→ Has-A Relationship is unidirectional (one way Relationship).

it means "Car Has-A Engine". but we can't say "Engine Has-A Car".

→ Composition : When one object is strongly Associated with another object (it means without Relationship they are not useful), then it called composition.

Ex→

→ Aggregation : when one object is weakly associated with another object (it means without Relationship both object have there own uses), then it is known as Aggregation.

Polymorphism :-

→ in java polymorphism is mainly divided into two types-

Polymorphism

(i) Compile-time polymorphism
or Static Binding
or Early Binding
↓
to achieve
↓
Method Overloading
(Function overloading)

(ii) Runtime polymorphism
or Dynamic Binding
or Late Binding.
↓
to achieve
↓
Method Overriding
(Function Overriding).

* Method Overloading :-

→ When there are multiple functions with the same name but different parameters then these functions are said to be overloaded.

→ Methods can be overloaded by changes in the number of arguments or/and a change in the datatype of paramenters.

→ we can create n number of functions with same name in a class but parameters or datatype of parameters or sequence of parameters must be different for each function.

→ Same name methods lies in same class or inherited classes with different parameter.

→ Inheritence is no compulsory.

→ class Demo {

```
    static int add(int a, int b){
        return (a+b);
    }
    static int add(int a){
        return (a+b);
    }
    static int add(int a, int b, int c){
        return (a+b+c);
    }
    PSVM(String [] args){
        Sop(add(4));
        Sop(add(4, 9));
        Sop(add(4, 4, 9));
    }
}
```

→ here, a function add is present in three places with same name but on each place add has different behaviour (work), that's why we can say here polymorphism is acheiving.
and it is deciceding at compile time that who's function going to be used, that's why we can say here also Runtime Polymorphism achieved.

```
class Parent{
    int num=10;
    void show(){
        SOP("Inside show in Parent");
    }
}
class Child extends Parent{
    int num=20;
    @Override
    void show(){
        SOP("Inside show in child");
    }
    PSVM(String args[]){
        Parent obj = new Child();
        SOP (obj.num);  -> 10
        obj.show();  -> Inside show in child.
    }
}
```

→ here, before calling obj.show(); compiler goes on parent and checking that is show() is present in parent or not, so, we can say here compile time polymorphism achieved.

but in runtime it will checked it who's object is present in obj then it will call the method of that class but data members are called of parent class.

it means here we can say it is deciding on runtime that who's method will be run so, we can say here runtime polymorphism achieved.

Interface :-

→ Interface is a blue print of class (behaviours).

→ A java interface contains static constants and abstract methods.

→ Interface is mainly used to achieve full abstraction.

→ Interface is mainly used to achieve the functionality of multiple Inheritence.

→ Interface is used for achieving loose coupling.

→ data members Inside interface are by default, public final static and member functions are by default, public abstract.

→ If you want to make an interface then in that case we need interface keyword followed by interface name.

→ classes can implements an interface where interface can extends an interface.

→ a single class can implement multiple interfaces. ~~where~~

→ an Interface can extends ~~more~~ multiple Interface.

→ an Interface can not extends a class.

→ Interface can not be initiated which means we can not create object of interface.

↳ because interface contains abstract methods and interface is not a class.

→ From Java 8 we can create concrete methods in interface with the help of default and static keyword.

→ we can create private methods in interface after (from) JDK 9.

→ Nested Interface is possible.

→ It is not necessary to create abstract methods in interface.

→

→ We can't create object but we can create reference of an interface.

→ a class which implementing Interfaces it must have to define all abstract methods that are present in interfaces.

→ if we want to implement some methods of interface (not all) then must have to make class as an abstract.

→ if we want to write the definition of a abstract method then we must have to write Access modifier same as abstract declaration part.

→ if we want to use the concrete methods and data members of ~~abstract~~ interface then we have to make Interface type reference variable but in this variable we have to make an object of class that implementing this interface. (upcasting)

→ we can't create constructor of Interface.

→ if we want to implement an interface and extends a class in a single class then we have to first extend then implement.

→ we can implement an interface in abstract class.

→ we can implement and extend both in a class at a time.

→ if we implements two interfaces in a single class and both interfaces have same name member then if we want to access members in class then we can access members with the help of <Interface name>.<member name>

→ Interface is not a class.

→ interface provides 100% abstraction before JDK 8 but after JDK 8 we can also create concrete methods inside ~~abstract~~ interface that's why now days we can achieve 0 to 100% abstraction by using interface.

→ If we want to make private method in interface then this method must be static after JDK 1.9

* we can use two or more access modifiers for same member at a time.

public default void show() { };

→
```
interface Demo{
        void show();
        public static display(){
                SOP("display called");
        }
}
class Main implements Demo {
        public show(){
                SOP("show called");
        }
        PSVM(String args[]){
                Main m = new Main();
                Demo d = new Main();
                m.show(); → show called
                d.show(); → show called
                d.display(); → display called.
        }
}
```

* Functional Interface :-

A functional interface is an interface that contains only one abstract method.

→ From java 8 lambda expression can be used to represent the instance of a functional interface.

→ to represent the functional interface we can use @FunctionalInterface annotation.

→ Functional Interface is additionally recognized as a Single Abstract Method Interface. In short, they are also known as SAM interfaces.

→ Some Built-in Java Functional Interfaces—

- Runnable → This interface only contains the run() method.
- Comparable → This interface only contains the compareTo()
- ActionListener → contains actionPerformed()
- Callable → This contains call()

* Marker Interface :-

It is an empty interface (no field or method).

Eg → Serializable, Clonable, etc.

this are the Marker interface.

FUNCTIONAL PROGRAMMING

→ Functional Programming focuses on what needs to be done rather than how it can be done.

→ Functional programming is a paradigm or pattern which allows programming using expression.

→ Functional programming works on only functional Interfaces.

→ Functional programming is the feature of java 8. after java 8 functional programming is allowed in java.

→ java is facing a problem of boiler plate code means the same code written multiple time with low or little modification.

→ Advantages of functional Programming:

1) Concise and Expressive code
2) Modularity and reusability
3) Easier Debugging
4) ~~code reusability~~

→ Functional programming consist of
1. Lambda Expression
2. Method Reference
3. Predicate

(1.) Lambda Expression :-

→ Lambda Expression is an anonymous function and it does not belongs to any class.
→ Lambda Expression completes incomplete methods with the help of expression.
→ Lambda expression to overcome the anonymous class.
→ Lambda expression works on only functional interfaces.
→ Lambda expression replace the anonymous class.
→ we can not use lambda expression outside of a method.

(1) <interface_name><ref-var-name> = ( ) → { } ;
or <abs_class_name>

Argument accepter. | Arrow tocken | method body part.

return

(2) <interface_name><ref-var-name> = (<parameters>) → (<operation>) ;
<abs_class_name>

or → {<code block>};
or → {<code block>
return ; };
or → SOP ("Hello") ;

→ we can also return the values from lambda expression.
→ it is not necessary to write datatype of parameter in Lambda expression if in outer scope have not any variable with same name.

→ if there is only one statement in lambda expression you may or may not use ~~return~~ return keyword.
You must use return keyword when lambda expression contains multiple statements. (for retur value).

→ lambda expression implement only the abstract function

→
```
@FunctionalInterface
interface Drawable{
        public void draw();
}
public class LambdaDemo{
        public static void main(String[] args){
                int num = 10;
                Drawable d2 = () ->{
                        System.out.print(" Hello num is : "+ num);
                };
                d2.draw();
        }
}
```

→
```
@FunctionalInterface
interface Addable{
        int add(int a, int b);
}
public class LambdaExp{
        public static void main(String[] args){
                Addable add1 = (a,b) -> (a+b);
                SOP(add1.add(10, 20)); - 30
                int c = add1.add(100, 200);
                SOP(c); - 300
        }
}
```

(2.) Method Reference :-

→ it is compact form of lambda expression

→ in method reference we provide reference of prewritten method to incomplete abstract method.

→ ~~there~~ it works on only functional interfaces.

→ there are 3 types of method References are :

(1) Reference to a static method

(2) Reference to an instance method

(3) Reference to a constructor

(1) Reference to a static Method:

you can refer to a static method defined in the class or interface. with following syntax:

<interface_name><ref_var_name> = <ref_class_name> :: <ref_fun_name>

or

<interface_name><ref_var_name> = <ref_interface_name> :: <ref_fun_name>

→ reference function (method) must be static in reference class or reference interface.

→ ~~class~~

```
interface Sayable {
        void say();
}
public class MethodRef {
        public static void main(String[] args){
                SOP("Hello, it's main");
                Sayable sayobj = MethodRef :: saySome;
                sayobj.say();  - Hello, I an inside static.
        }
        public static void saysome(){
```

```
        SOP(" Hello, I am inside static ");
    }
}
```

→ @FunctionalInterface

```
interface Operation {
        void operation();
        Static void myfun(){
            SOP(" Inside myfun ");
        }
}
Class MethodRef {
    public static void main(String[] args){
        Operation obj = Operation :: myfun;
        obj.operation(); — Inside myfun.
    }
}
```

(2) Reference to an Instance Method:

like static methods you can also refer to a instance method. with following Syntax:

<interface-name> <ref-var-name> = <ref-object-name> :: <ref-method-name>

→ ~~method~~ referenced method must be non-static in reference object class.

→ @FunctionalInterface

```
interface Sayable {
        void say();
}
```

```
class MethodRef{
    public static void main(String[] args){
        MethodRef mobj = new MethodRef();
        Sayable sayobj = mobj :: saysome();
        sayobj.say();  — Inside saysome.
    }
    void saysome(){
        SOP("Inside Saysome");
    }
}
```

(3) Reference to a Constructor:

we can refer to a constructor method by using new keyword. by using following syntax:

<interface-name><ref-var-name> = <ref-class-name> :: new.

→ here new keyword represents that constructor who parameters of abstract method.

it means parameters of constructor = parameters of abstract method.

→
```
@FunctionalInterface
interface Operation{
    void operation(int a, int b);
}
class Reference{
    Reference(){
        SOP("default constructor");
    }
    Reference(int a, int b){
        SOP("sum:"+(a+b));
    }
```

```
public static void main(String[] args){
        Operation obj = Reference :: new;
        Obj.operation(3, 6); - Sum : 9
    }
}
```

(3.) Predicate :-

a 'Predicate' is a functional interface from 'java.util.function' package that takes an input and returns a boolean value.

→ the 'Predicate' interface has a single abstract method
Test(T t)
it also provides few default methods like
and
or
negate
isEqual

→ this all methods are return boolean value.

→ It is used for unit testing.

→ the method is used to test if a given object satisfies a particular condition.

→ java.util.function.Predicate
this is the fully qualified name of Predicate.

```
import java.util.function.Predicate;
class PredicateDL {
    public static void main(String[] args){
    Predicate<Integer> lessThan = i -> i<=100;
    SOP("Result : " + lessThan.test(45));  -> true
    SOP("Result : " + lessThan.test(145));  -> false
    SOP("Result : " + lessThan.negate().test(45));  -> false

    Predicate<String> str = Predicate.isEqual("Andrew");
    SOP("Result : " + str.test("Andrew"));  -> true
    SOP("Result : " + str.test("andrew"));  -> false

    Predicate<Integer> lessThan1 = i -> i<=100;
    Predicate<Integer> greaterThan1 = i -> i>=100;

    SOP("Result : " + lessThan1.and(greaterThan1).test(75));  -> false
    SOP("Result : " + lessThan1.or(greaterThan1).test(110));  -> true

    }
}
```

Object Class :-

→ Object class is the parent class of every class in java.

→ Every class directly or Indirectly extends Object class.

→ Object class belongs inside default package java.lang

→ java.lang package is the default pkg in java

- → import java.lang.Object
- → import java.lang.String
- → import java.lang.System.

→
```
class Hello extends Object {   ← directly extends Object class
        psvm (String args[]){
        }
}
```

→
```
class Object {
}
class Parent extends Object {   ← directly extends Object class
}
class Child extends Parent {   ← indirectly extends Object class.
        Psvm (String args[]){
        }
}
```

→ Object class provides some built in methods:

(1) toString()
(2) hashCode()
(3) getClass()
(4) equals()
(5) finalize()
(7) wait()
  → wait(argument)
  → wait(1-argument, 2-argument)
(8) notify()
(9) notifyAll()
(6) clone()

} Garbage collector

} multithreading

} object cloning.

(1) toString() :-

→ toString() method provides default string Representation of an object.

→ we can Override toString() method.

→ default string representation of every object will be different.

```
class Student {
        private int id;
        private String email, password;
        public void setData (String email, String pass ,int id) {
            this.password = pass;
            this.email = email;
            this.id = id;
        }
        @Override
        public String toString() {
            return id + "\t" + email + "\t" + password;
        }
}
class Object {
        psvm (String args[]) {
            Student obj = new Student();
            obj.setData ("hemant@gmail.com", "hemant@123", 123);

            SOP (obj);
            SOP (obj.toString());
        }
}
```

O/P: 123 hemant@gmail.com hemant@123
123 hemant@gmail.com hemant@123

(2) hashcode():-

hashcode() method returns the hash code for the method class object.

→ Hashcode is a unique code generated by the JVM at time of creation of the object.

→ Hashcode: 1643976273

→ hashcod is integer value (code).

→

```
class Product {
    private int pid;
    private String name;
    @Override
    public int hashcode(){
        return pid;
    }
    public Product(int id, String name){
        pid=id;
        this.name=name;
    }
}
class Object2{
    psvm(String args[]){
        Product p1 = new Product(101, "soap");
        SOP(p1.hashcode());
    }
}
```

O/P: 101

(4) equals ( ) :-

→ this method is used to compare two objects.

→ It returns true if (this object is same as the object argument) both objects are same otherwis return false.

```
class Student {
        String name;
        Student (String name) {
            this.name = name;
        }
}
class Demo {
        psvm (String args[]) {
            Student s1 = new Student ("Hemant");
            SOP (s1. hashcode);
            SOP (s1. name);

            Student s2 = new Student ("Hemant");
            SOP (s2. hashcode);
            SOP (s2. name);

            SOP ( s1. equals (s2) );

            s2 = s1;
            SOP (s1. equals (s2));
        }
}
```

Output: 1295643221
Hemant

1295649224
Hemant

false
true

→ equals method check references of objects. & not values.

(3) getClass():-

→ getClass method returns the class ~~of~~ of object.

```
class Object {
    psvm (String args[]){
        Object obj = new String();

        SOP( obj.getClass()) ;
        sop( obj. getClass(). getName());
    }
}
```

Output:

```
class java.lang.String
java.lang.String
```

#. Package :-

Package is the ~~collection~~ machenism of the collection of classes and interfaces at one place.

→ Java package helps in avaiding name conflection. (it means we can not create duplicate name classes in a pkg).

→ Package is a group of similer type of classes, interfaces and sub-Packages.

→ there are two type of Packages -

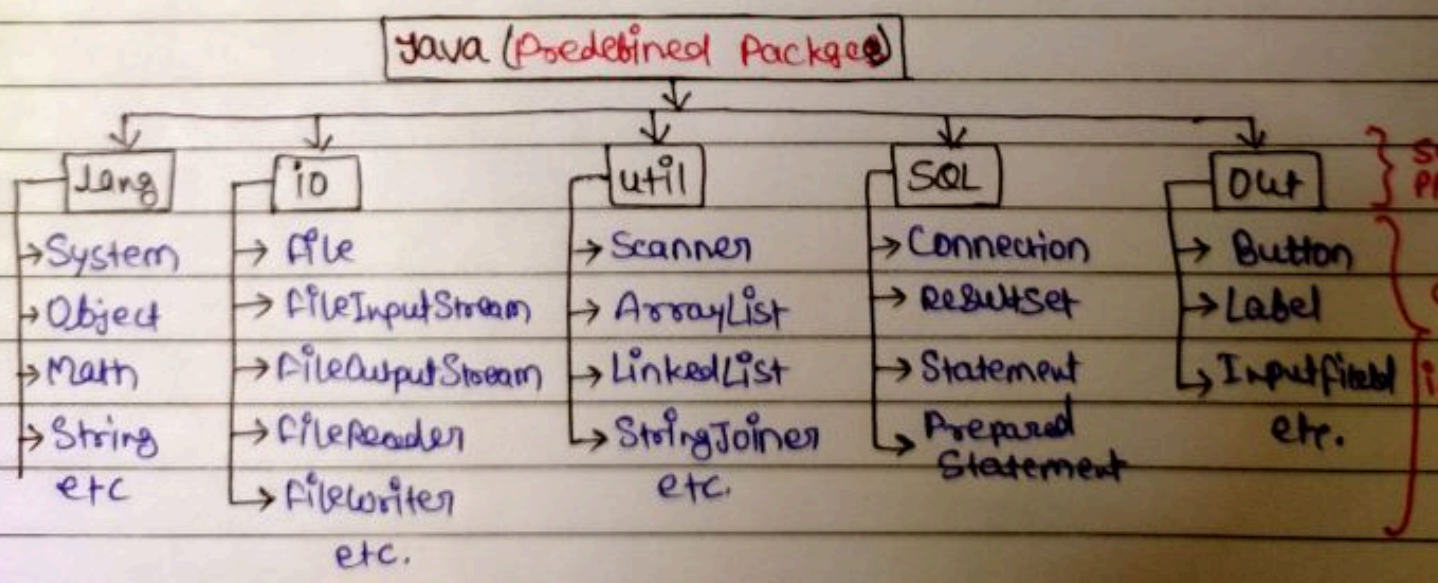
(i) Built-in Package :(Standard or Predefined Package)

(ii) user Defined Package.

→ (i) Built-in Package : Built-in packages consists of java API's which comes with java Runtime Environment.

Ex. → java.Util , java.io etc.

→ (ii) User Defined Package :- user defined packages ~~consists~~ are those packages which are created by user according to programming needs.



→ Java Packges Provides Access Protection.

→ if you import any Package, subpackges will not imported only classes and interfaces ~~of~~ of that class are imported.

→ Syntax for import Whole Packges (Excluding Sub-Pkges):

```
import <Pkg Name>.* ;
```

→ Syntax for import any ~~ete~~ perticular class or interface:

```
import <Pkg Name>.<class or interface name> ;
```

→ import < fully qualified name >;

→ Sub-Packge: Sub-Package is a package inside a packge. Sub-Package also contains classes and interfaces. and also sub-packages.

→ java Program compilation and Run command:

```
javac -d . <file Name>
java <PkgName>.<.classFileName>
```

→ package keyword is used to create a packge in java.

```
package mypack;
class{

}
```

→ How to send the .class file to another directory or drive?

```
javac -d <full Pathname of dir> <.java file name>
set classpath=<fll path name of .class file dir.>
java <PkgName>.<.classFilename>.
```

→ Static import :- Static import concept is introduced in 1.5 Version.

→ with the help of static import, we can access the static members of class directly without class name or any object.

Ex→ by using static import we can use sqrt() function with its class name 'Math'.

```
import static java.lang.Math;
import static java.lang.System.*;
class Test{
    psvm(String args[]){
        out.println(sqrt(4));  → 2
        out.println(pow(2,2));  → 4
    }
}
```

→ Package class :

The Package class provides methods to get information about the specification and implementation of a package.

→ It provides methods.

```
class Test{
    PSVM(String[] args){
        Package p = Package.getPackge("java.lang");
        sop("Pkg Nam :"+ p.getName());
        sop("Specification Title:"+ p.getSpecificationTitle());
        sop("Specification Vendor:"+p.getSpecificationVendor ());
        sop("Specification Version:"+p.getSpecificationVersion());

        sop("Is Saled :"+ p.issaled ());
        :
        :
    }
}
```

PkgName : java.lang
Specification Title : Java Platform API Specification
Specification Vendor : Sun Microsystem, Inc.
Specification version : 1.6
Is Saled : false.

String in Java :

String is a sequence of characters. in java string is an object which represents which represents sequence of characters.

→ String is a non-premitive data type in java.

→ in Java there are two types of Strings are -

(1) Mutable String

(2) Immutable String

(1) Immutable String :

→ Immutable Strings are strings that value can't be change.

→ we can create Immutable String using String class.

(2) Mutable String :

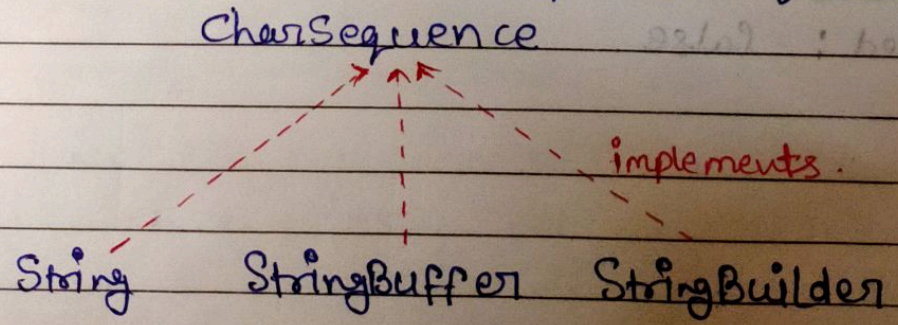
→ Mutable Strings are string that value can be change.

→ there are two way to create mutable String :

(1) by StringBuffer class

(2) by StringBuilder class (Java 1.5)

* CharSequence Interface : the CharSequence Interface is used to represent the sequence of characters.



String class : Immutable String :-

String is a sequence of characters. in java string is an object which represents sequence of characters.

→ String is a non-premitive datatype in java.

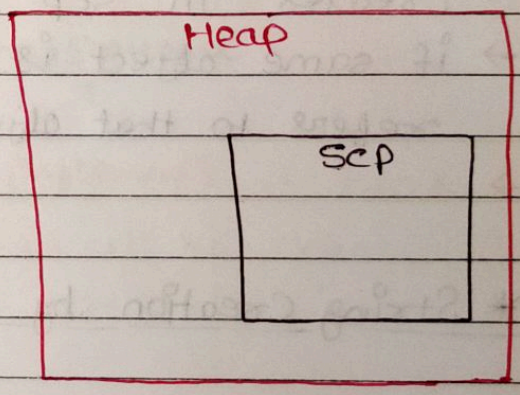
→ Strings are immutable, which means we can't change the value of String.

→ there are two ways to create object of String class:

(a) by String Literal : String name= "Hemant" ;

(b) by new keyword : String name = new String("Hemant");

* String Constant Pool : (SCP)



→ String Constant Pool is a storage area in Java heap where string literals stores.

→ it is also known as String Pool or String Intern pool.

→ by default it is empty and privately maintained by the Java String Class.

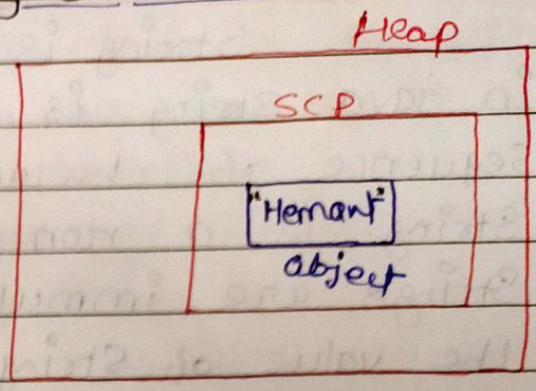
→ Garbage collector does not work in SCP.

→ when ever a new string (object) literal is going to store in scp, first it checks that if any object that has same value is present in scp or not, if object is present then it refers that object (new object is not created) otherwise it creates a new object in scp area.

→ Duplicate objects (that have same values) are not created in SCP.

* String Creation by ~~new~~ ~~keyword~~ String Literal :-

```
class Demo{
    psvm(String args[]){
        String nam = "Hemant";
    }
}
```



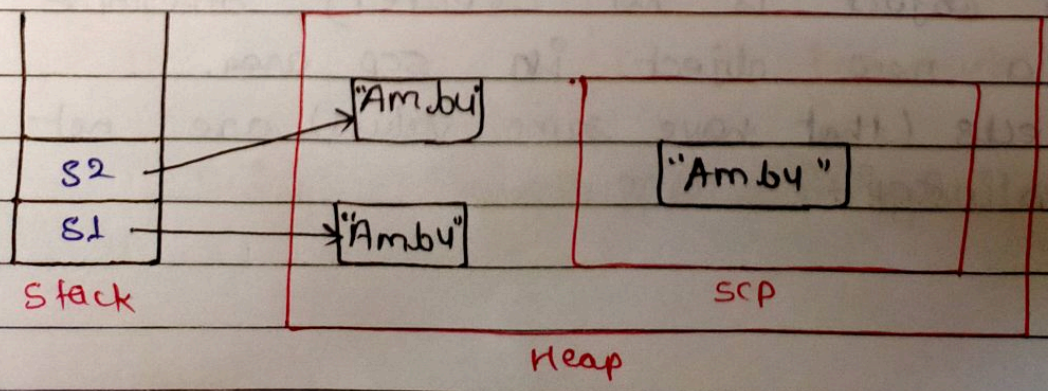
→ Whenever we create an String object by new keyword then it takes memory in scp. it means object is creates in scp not in heap.

→ if same object is allready present in scp then it refers to that object, do not new object.

→

* String Creation by new keyword :-

```
class Demo{
    psvm(String args[]){
        String s1 = new String("Amby");
        char ch[] = {'A','m','b','u'};
        String s2 = new String(ch);
    }
}
```

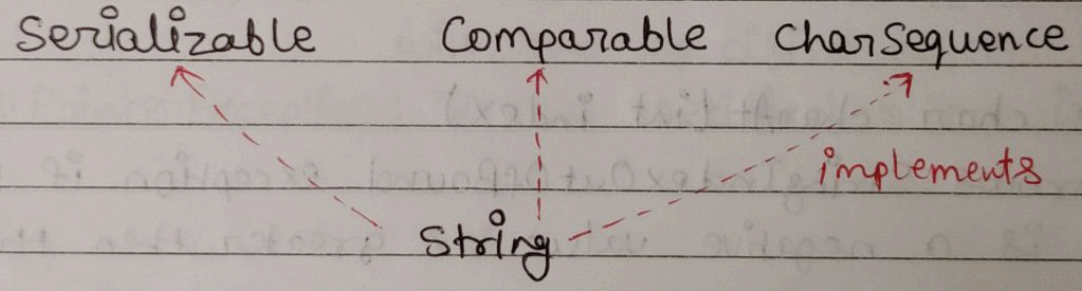


→ Whenever we create an object of String by new Keyword then two objects are created first one is created inside Heap (outside SCP) and ~~sero~~ second is created inside SCP.

→ but Reference variable refer to the object that is created inside heap (outside SCP).

→ We can access the object that is created inside SCP using intern() method.

* String class implements some ~~methods~~ interfaces;



* Why Java Strings are Immutable ?

* String methods :-

there are many methods are avilable to perform the operations on Strings. Some are following:

(1) charAt () :-

the Java String class charAt () method returns a char value at the given index number.

```
String name = "Hemant";
char ch = charAt (2);
                  ↳ index
sop (ch) ; → 'm'
```

- public char charAt (int index)
- It throws StringIndexOutOfBound exception if the index is a negative value or greater then this String length.

(2) compareTo () :-

→ the Java String class compareTo () method compares the given String with the current String lexicographically.

→ it returns a number (Positive, negative, or zero).

→ it compares Strings on the basis of the unicode value of each character in the strings.

→ If the first string is lexicographically greater then the second string, it returns a positive number (difference of character value).

if S1 > S2, it returns positive number.

NOTe:
* == Operator compares references not values.

→ If the first string is less then the second string lexicographically, it returns a negative number.
if S1 < S2, it returns negative number.

→ If the first string is lexicographically equal to the second string, it return 0.
if S1 == S2, it returns 0.

→ public int compareTo(String str)
↳ string parameter.

→ It can throw the following two exceptions:
(1) ClassCastException : If this object cannot get compared with the specified object.
(2) NullPointerException : If the specified object is null.

→
```
String S1 = "hello";
String S2 = "hello";
String S3 = "meklo";
String S4 = "hemlo";
String S5 = "flag";
```
SOP(S1.compareTo(S2)); → 0
SOP(S1.compareTo(S3)); → -5 (because 'h' is 5 time less then 'm')
SOP(S1.compareTo(S4)); → -1 (because 'l' is 1 time lower then 'm')
SOP(S1.compareTo(S5)); → 2 (because 'h' is 2 time greater then 'f')

→ when we compare two string in which either first or second string is empty, the methods returns the difference of 'length of strings'.

NOTE:
* We can concat strings using + operator.

(3) concat() :-

→ the Java String class concat() method combines specified string at the end of the string.
→ it returns a combined string.
→ public String concat (String str)
↳ String parameter.

→ String s1 = "Hemant";
String s2 = "Prajapati";
String s3 = " ";
String s4 = s1.concat(s2); → HemantPrajapati
String s5 = s1.concat(s3).concat(s2); → Hemant Prajapati.

(4) contains():

→ the Java String class contains() methods search the sequence of characters in this ~~Stra~~ String.
→ if the sequence of values is found, it returns true otherwise returns false.
→ public boolean contains (CharSequence sequence)
→ it can throw NullPointerException if the sequence is null.
→

String name = "My name is Hemant";
sop(name.contains("name"); → true
sop(name.contains("hello"); → false
sop(name.contains("Hemant"); → true
sop(name.contains("hemant"); → false.

(5) endsWith( ) :-

→ the Java String class endsWith() method checks if this string ends with a given suffix or not.

→ if string ends with given suffix then the method returns true otherwise method returns false.

→ public boolean endsWith(String suffix)

→ String name = "Hemant Prajapati";
SOP(name.endsWith("ti"); → true
SOP(name.endsWith("i"); → true
SOP(name.endsWith("t"); → false

(6) equals() :-

→ the Java String class equals() method compares the two given strings based on the content of the String.

→ if all characters are matched then it returns true. otherwise it returns false.

→ public boolean equals(Object obj)

→ String name1 = "Hemant";
String name2 = "Hemant";
String name3 = "hemant";
SOP(name1.equals(name2)); → true
SOP(name2.equals(name3)); → false.

(7) equalsIgnoreCase() :-

→ the Java String class equalsIgnoreCase() method compares the contents of two string irrespective of the case of the String.

→ public boolean equalsIgnoreCase(String str)

→ if the content of both Strings (ignoring case) are same then method returns true otherwise returns false.

→ String name = "Hemant";
String temp = "hemant";
SOP(name.equalsIgnoreCase(temp)); → true.

(8) length():-

→ the Java String class length() method finds the length of String.

→ public int length()

→ returns the length (size) of string.

→ the length of Java String is the same as the unicode units of the String.

→ String s1 = "Hello";
SOP(s1.length()); → 5

(9) toLowerCase():-

→ the Java String class toLowerCase() method returns the string in lower case letter.

* → the toLowerCase() method works same as toLowerCase(Local.getDefault()) method. It internally uses the default locale.

→ public String toLowerCase()

→ public String toLowerCase(Locale locale)

→ String name1 = "HemAnT";
String name2 = "HEMANT";
SOP(name1.toLowerCase()); → hemant
SOP(name1.toUpperCase()); → HEMANT
Char ch[] = name2.toCharArray();
for(int i=0; i<ch.length; i++){
SOP(ch[i]);
}

(10) toUpperCase() :-

→ the Java String class toUpperCase() method returns the string in upper case letter.

→ public String toUpperCase()

→ public String toUpperCase(Locale locale)

(11) toCharArray() :-

→ the Java String class toCharArray() method converts this string into char Array.

→ it returns newly created character Array.

(12) format() :-

→ the java string class format() method returns the formatted String by given locale, formate and arguments.

→ If you don't specify the locale in String.format() method, it uses default by calling Locale.getDefault() method.

→ public static String format(String format, Object... args)

→ public static String format(Locale locale, String format, Object... args)

→ it can throw NullPointerException if format is null.

→ it can throw IllegalFormatException if format is illegal.

```
String name = "Hemant";
String s1 = String.format("Name is : %s", name);
String s2 = String.format("s2 is %f", 32.3467);
String s3 = String.format("s3 is %.3f", 32.06579);
SOP(s1); → Name is Hemant
SOP(s2); → s2 is 32.3467
SOP(s3); → s3 is 32.065.
```

(13) getBytes():-

→ the java String class getBytes() method does the encoding of String into the sequence of bytes and keeps it in an array of bytes.

→ it returns byte array.

→ public byte[] getBytes()

→ public byte[] getBytes(~~Character~~ Charset ch)

→ public byte[] getBytes(String str) throws UnsupportedEncodingException.

→ it can throw UnsupportedEncodingException when the mentioned charset is not supported by the method.

```
String s1 = "ABCDEFG";
byte[] ba = s1.getBytes();
```
→

65	66	67	68	69	70	71

(14) getChars():-

→ the Java String class getChars() method copies the content of this string into a specified char array.

→ public void getChars(int srcBeginIndex, int srcEndIndex, char[] destination, int dstBeginIndex)

```
→ String str = "hellow user how are you?";
  char[] ch = new char[10];
  try{
      str.getChars(7, 18, ch, 0);
      SOP (ch);
  }
  catch(Exception e){

  }
```
→ user how are

(15) indexOf():-

→ the Java String class indexOf() method returns the position of the first occurance of the specified character or String in a specific String.

→ int indexOf(int ch)
it returns the index position for the given char value.

→ int indexOf(int ch, int fromIndex)
it returns the index position for given char and from index

→ int indexOf(String substr)
it returns the index position for the given substring.

→ int indexOf(String substr, int fromIndex)
it returns the index position for the given substring and from index.

```
String s1 = "this is index of example";
int index1 = s1.indexOf('s'); → 3
int index2 = s1.indexOf('s', 4); → 6
int index3 = s1.indexOf("index"); → 8
int index4 = s1.indexOf("is", 4); → 5
```

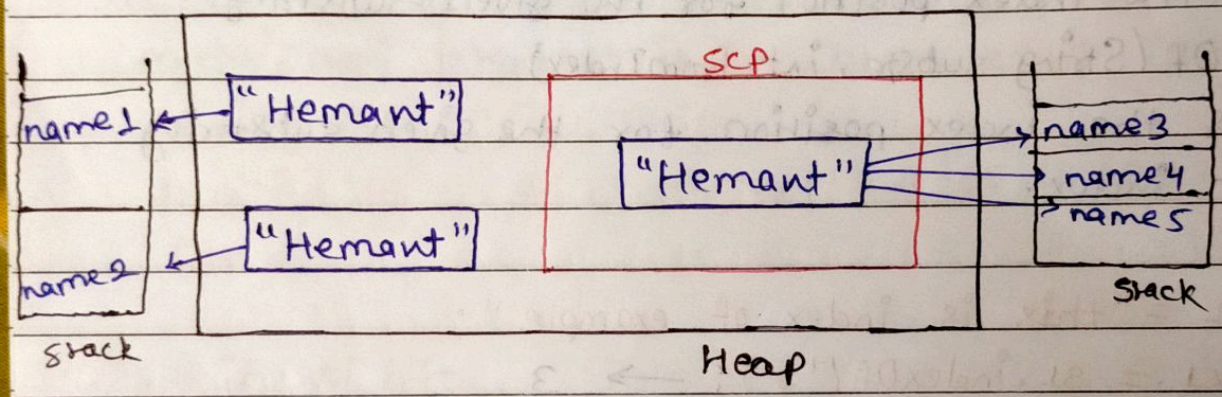
(16) intern():-

→ The Java String class intern() method represent (return) the String from (of) SCP.

→ when we make String using new keyword then two object are created first one is on heap and second is in SCP. by default reference variable refers to heap area String but when we use intern() method it refers to SCP String.

→ public String intern().

```
String name1 = new String("Hemant");
String name2 = new String("Hemant");
SOP(name1 == name2); → false
String name3 = name1.intern();
String name4 = name2.intern();
SOP(name3 == name4); → true
String name5 = "Hemant";
SOP(name3 == name5); → true
SOP(name4 == name5); → true.
```



(17) isEmpty():- (Java 1.6)

→ the Java String class isEmpty() method checks if the input String is empty or not.

→ if empty then returns true otherwise return false.

→ if String str = null;
then str.isEmpty throws exception.

→ public boolean isEmpty()

```
String str1 = "";
String str2 = "Hemant";
SOP(str1.isEmpty()); → true
SOP(str2.isEmpty()); → false
String s = null;
SOP(s.isEmpty()); → throws NullPointerException.
```

(18) join() :-

→ the Java String class join() method returns a String joined with a given delimiter.

→ public static String join(CharSequence delimiter, CharSequence... elements)

→ public static String join(CharSequence delimeter, Iterable<? extends CharSequence> elements)

→ it can throw NullPointerException when element or delimeter is null.

```
String str = String.join("-","Welcome","to","India");
SOP(str);→ Welcome-to-India
```

(19) lastIndexOf():-

→ the Java String class lastIndexOf() method returns the last index of the given substring or character value.

→ int lastIndexOf(int ch)
It returns last index position for the given char value

→ int lastIndexOf(int ch, int fromIndex)
It returns last index position for the given char value and from index.

→ int lastIndexOf(String substr)
It returns last index position for given substring

→ int lastIndexOf(String substr, int fromIndex)
It returns last index position ~~from~~ for given substring and from index.

```
String s1 = "this is index. of example";
int index1 = s1.lastIndexOf('s'); → 6
int index2 = s1.lastIndexOf('s', 5); → 3
int index3 = s1.lastIndexOf("of"); → 14
int index4 = s1.lastIndexOf("of", 19); → 14
int index5 = s1.lastIndexOf("of", 10); → -1
```

(20) replace():- (JDK 1.5)

→ the Java String class replace() method returns a string replacing all the old char or CharSequence to a new char or CharSequence.

→ public String replace(char oldchar, char newChar)

→ public String replace(CharSequence target, CharSequence replacement)

→ it can throw NullPointerException if target or replacement is null.

```
String s1 = "Hemant is bad boy";
String s2 = s1.replace("bad", "good");
String s3 = s1.replace('H', 'h');
SOP(s2); → Hemant is good boy
SOP(s3); → hemant is bad boy.
```

(21) replaceAll():-

→ the java String class replaceAll() method returns a String replacing all the sequence of characters matching regex and replacement string.

→ public String replaceAll(String regex, String replacement)

→ it can throw PatternSyntaxException if the syntax of the regular expression is not valid.

```
String s1 = "Hello Dosto Mai Hu Doremon";
SOP(s.replaceAll("o", "a"); → Hella Dasta Mai Hu Daremon.
```

(22) split () :- (Java 1.4)

→ the java String split() methods splits this String against given regular expression and returns a character array.

→ public String split (String regex)

→ public String split (String regex, int limit)

→ it can throw PatternSyntaxException if pattern for regular expression is invalid.

```
String s1 = "I am happy";
String word1[] = s1.split ("a");
```

→ | "i" | "m h" | "ppy" |

(23) startsWith():-

→ the Java String class Strartswith() method checks if this string starts with the given prefix or not.

→ if starts with given prefix then return true o/w false.

→ public boolean startsWith(String prefix)

→ public boolean startsWith(String prefix, int offset)

```
String str = "Hello Hemant how are you";
sop(str.startsWith("He"));  → true
sop(str.startsWith("llo",2));  → true
sop(str.startsWith("how"));  → false
```

(24) trim():-

→ the Java String class trim() method removes space from before and after String.

→ public String trim()

→ returns String with omitted leading and trailing spaces.

```
String str1 = " Hello";
String str2 = " Doremon";
sop(str1.trim() + str2);   Hello Doremon
sop(str1.trim() + str2.trim());   HelloDoremon
String str3 = "Nobita ";
sop(str3.trim());   Nobita
```

(25) valueOf():-

→ valueOf method converts different type of values into string.

→ it returns String representation of given value.

→ public static String valueOf(boolean b)

→ public static String valueOf(char ch)

→ public static String valueOf(int i)

→ public static String valueOf(char[] c)

→ public static ~~void~~ String ~~(long l)~~ valueOf (long l)
→ public static String ~~(float f)~~ valueOf(float f)
→ public static String valueOf(double d)
→ public static String valueOf(Object o)

SOP(String.valueOf('A')); → A
SOP(String.valueOf(69)); → 69

(26) substring():-

→ substring() method returns part of string.
→ public String substring(int startindex)
→ public String substring(int startindex, int endindex)
→ it can throw StringIndexOutOfBoundsException.

(27) compareToIgnoreCase():-

→ It works as compareTo() method Ignoring case.
→ public int compareToIgnoreCase(String str)

(28) contentEquals():-

→ contentEquals() method checks both string have same content or not. if yes then returns true o/w false.
→ public boolean contentEquals(CharSequence ch)
→ public boolean contentEquals(StringBuffer chars)

String str1 = "Hello Hemant";
str2 = "hello hemant";
SOP(str1.substring(6)); → Hemant
SOP(str1.substring(0,5)); → Hello
SOP(str1.compareToIgnoreCase(str2)); →
SOP(str1.contentEquals("Hello Hemant")); → true.

* Java StringJoiner class :- (Java 8)

→ Contained in Java.util package.

→ it is used to ~~construch~~ construct a sequence of characters separated by a delimiter.

→ Public StringJoiner(CharSequence delimiter)

→ public StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix):

→ constructors.

→ public StringJoiner add(CharSequence newElement)

→ public StringJoiner merge(StringJoiner other)

→ public StringJoiner setEmptyValue(CharSequence emptyValue)

→ methods.

*→
```
StringJoiner names = new StringJoiner(",");
names.add("Rahul");
names.add("ambu");
names.add("vaishnavi");
sop(names);
```
→ Rahul, ambu, vaishnavi

*→
```
StringJoiner hnames = new StringJoiner(",","[","]");
hnames.add("Rahul");
hnames.add("ambu");
hnames.add("vaishnavi");
sop(hnames);
```
→ [Rahul, ambu, vaishnavi]

*
```
StringJoiner names = new StringJoiner(",");
names.setEmptyValue("Hello");
sop(names)
```
→ Hello.

*→ StringJoiner names1 = new StringJoiner(";","[","]")

names1.add("ambu");

names1.add("vaishnavi");

StringJoiner names2 = new StringJoiner(":","[","]").

names2.add("hemant");

names2.add("vaishnavi");

sop(names1) → [ambu, vaishnavi]

sop(names2) → [hemant : vaishnavi]

StringJoiner merge = ~~new StringJoiner~~ names1.merge(names2);

sop(merge); → [ambu, vaishnavi, hemant : vaishnavi]

* String Remaining *

* Java StringTokenizer class :-

→ it contained in Java.util package.
→ StringTokenizer class allows you to break a String into tokens.
→ It is ~~bag~~ legacy class of Java.
→ StringTokenizer(String str)
→ StringTokenizer(String str, String delimeter)
→ StringTokenizer(String str, String delim, boolean returnvalue)
→ Constructors.

→ boolean hasMoreTokens() → checks if there is more token avilable
→ String nextToken() → returns the next Token
→ String nextElement() → returns the next token based on delimeter.
→ boolean hasMoreElement() → same as hasMoreTokens()
→ Object nextElement() → same as nextToken(); but return type Object.
→ int countTokens() → returns total no. of tokens
→ Methods.

Eg →
```
import java.util.StringTokenizer;
public class Simple {
    public static void main(String args[]){
        StringTokenizer st = new StringTokenizer("my name is
                                                 hemant", " ");
        while(st.hasMoreTokens()){
            sop(st.nextToken());
        }
    }
}
```

o/p: my
name
is
hemant.

Mutable String:-

→ mutable String means we can change the value of this String.

→ In java there are two ways to create mutable String —
① by StringBuffer class
② by StringBuilder class.

① StringBuffer class:-

→ StringBuffer class is used to create mutable String object.

→ StringBuffer() → creates empty String Buffer with default capacity of 16.
→ StringBuffer(String str) → creates a String Buffer
→ StringBuffer(int capacity) → creates an empty String Buffer with specified capacity as length.
Constructors.

→ public synchronized StringBuffer append(String s)
→ public synchronized StringBuffer insert(int offset, String s)
→ public synchronized StringBuffer replace(int startIndex, int endIndex, String str).
→ public synchronized StringBuffer delete(int startIndex, int endIndex)
→ public synchronized StringBuffer reverse()
→ public int capacity()
→ public char charAt(int index)
→ public int length
→ public String substring(int beginIndex)
→ public String substring(int beginIndex, int endIndex)
→ public void ensureCapacity(int minimumCapacity).

→ Methods.

```
*class Demo{
    public static void main(String args[]){
        StringBuffer sb = new StringBuffer("Hello");
        sb.insert(1,"Java");
        sop(sb); → HJavaello
    }
}

*class Demo2{
    PSVM(String args[]){
        StringBuffer sb = new StringBuffer("Hello");
        sb.delete(1,3);
        sop(sb); → Hlo
    }
}

*class Demo3{
    PSVM(String args[]){
        StringBuffer sb = new StringBuffer("Hello");
        sb.reverse();
        sop(sb); → olleH
        sop(sb.capacity()); → 16
        sb.append("user");
        sop(sb); → olleH user
    }
}
```

(2) StringBuilder class:-

Java StringBuilder class is used to create mutable String.

→ the Java StringBuilder class is same as String Buffer class except that it is non-synchronized.

→ It is avilable since JDK 1.5

→ all constructors and methods are same as StringBuffer but it is non-synchronized.

* Difference Between StringBuffer and StringBuilder :-

StringBuffer	StringBuilder
① StringBuffer is synchronized.	① StringBuilder is non-synchronized.
② StringBuffer was introduced in Java 1.0	② StringBuilder was Introduced in Java 1.5
③ StringBuffer is less efficient than StringBuilder.	③ StringBuilder is more efficient than StringBuffer.

#. Object Cloning :-

→ Object cloning refers to the creation of an exact copy of an object

→ The clone() method of Object class is used to clone an object.

→ It creates a new ~~copy~~ instance of the 'class of the current Object' and initializes all its fields with 'Exactly the contents of the ~~coor~~ corresponding fields of this object'.

→ The clone() method is declared in object class.

protected native object clone() throws CloneNotSupportedException;

→ clone() method is a native method and it's definition part is not present in Object class.

→ the java.lang.Clonable interface must be implemented by the class whose object clone we want to create.

→ If we don't implement clonable interface, then clone() method generates CloneNotSupportedException.

→ Clonable interface is a marker interface.

→ Clonable interface simply gives the instruction to the compiler of that 'we want to make clone object of this class'.

Exception Handeling in Java:-

→ Exception's are the runtime anomalies (unexpected Situation) which may arise during the execution of program and the machenism of Handeling ~~Exceptions~~ such type of exceptions are known as Exception Handeling Machenism.

→ Exception Handeling Machenism:
1) Find the problem (Hit the Exception).
2) Inform that error has occured (Throw the exception).
3) Recieve error Information (catch the exception).
4) Take corrective ~~measures~~ measures (Handle the exception).

→ keywords used in Exception Handeling:
1) Try (for making try block)
2) Catch (for making catch block)
3) finally (for making finally block)
4) throw
5) throws

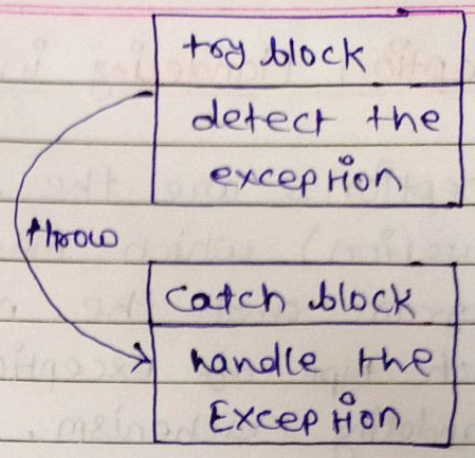
→ Difference Between Exception and Error:-

① In most of cases Exception occured by our program but Errors are occured because of lack of System Resources; not by our program & thus programmer can't do anything.

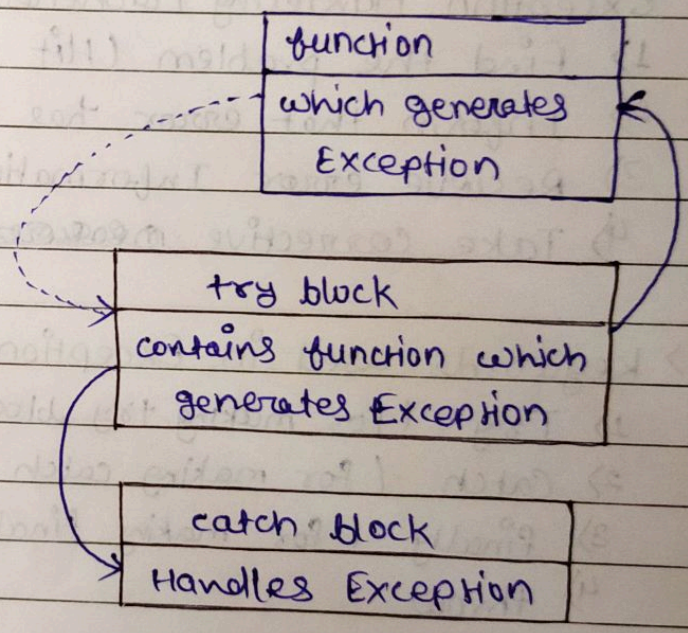
② Exceptions are recoverable. but Errors are not recoverable.

③

(1)
```
try {
    ------
    ------
    ------
}
catch( Exception e){
    ------
    ------
}
```



(2)
```
<return_type><function_name>( )
{
}
try{
    <function_name>();
}
catch(Exception e){
}
```



(3) ~~try~~ try with multiple catch
```
try{
}
catch(Exception){
}
catch( ——— ){
}
---------
---------
```

④ try with generalized catch

```
try {

}
catch(Exception name | Exception name  e ) {

}
```

→ pipe

⑤ try catch with finally

```
try{

}
catch(Exception e){

}
finally{

}
```

⑥ try with finally

```
try{

}
finally{

}
```

Note :

→ There are many acceptions in core and Advance java.

→ while using pipeing we can't use parent and child Exception together.

→ try block contains the code that may throw the exception. if any exception is occur's while executing try block then catch block catches the Exception and executes code written inside catch block.
finally block contains the code that is mendatory to execute even through exception occurs or not.

→ we can't use catch without try block.

→ we can't use finally without try block.

→ we can use try block without catch or finally or both.

→ userdefined Exception is checked Exception.

→ we can used checked and unchecked Exception together if there is any need.

→ Exception Handeling provides the alternet way to solve unwanted/unexpected situation/event rather then solving/repairing current event.

→ we can also use nested try-catch-finally.

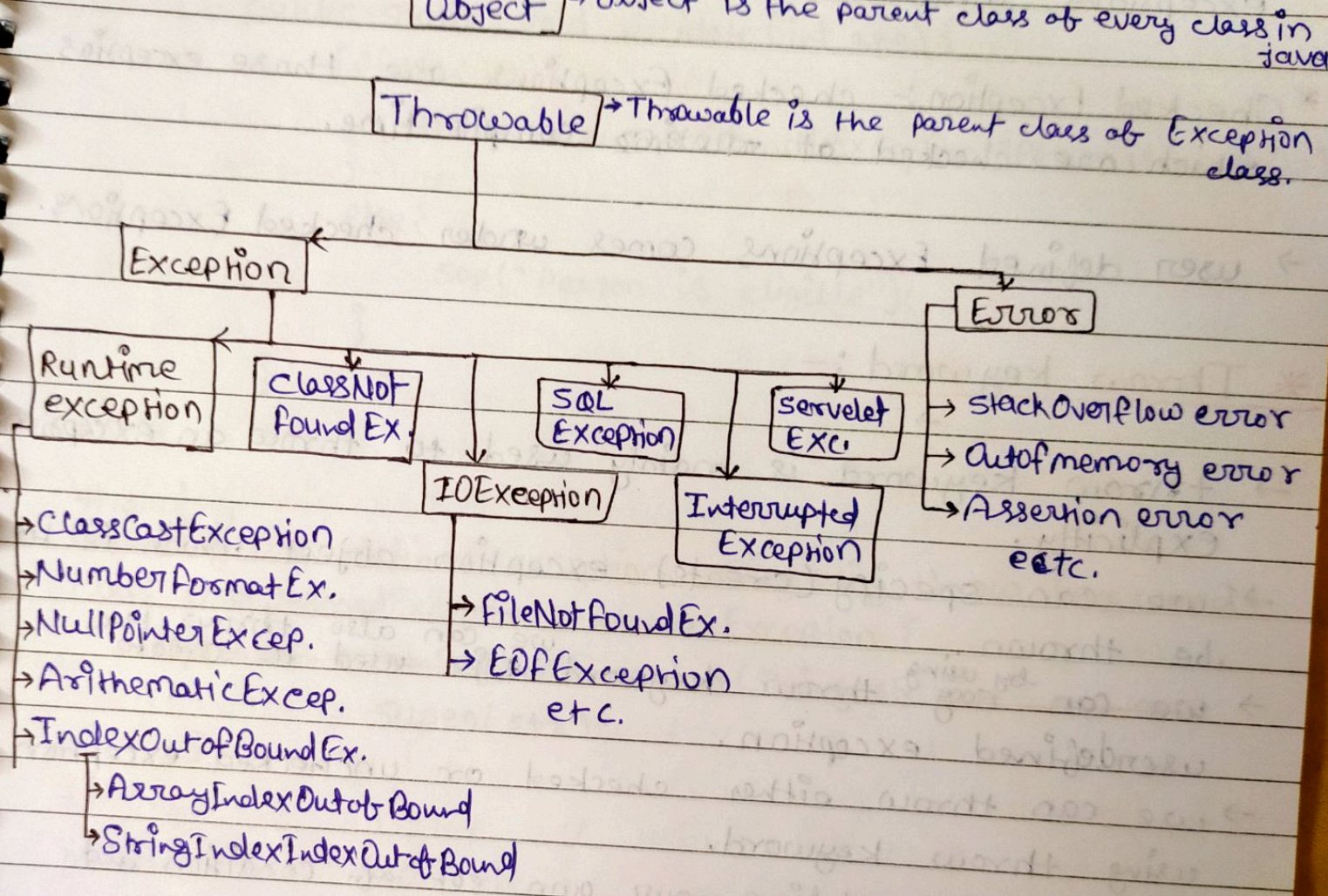
→ If we are calling a method that declares a exception

→ whenever there is exception, the method in which exception arise will create an object and that object will store this three things-

① Exception name ② description ③ Stack Trace.

→

* Heirarchy of Exception and Error



* Unchecked Exceptions :- Uncheacked Exceptions are those exceptions which arriesed at runtime.

* Checked Exception :- checked Exceptions are those exceptions which are checked at ~~runtime~~ compile time.

→ user defined Exceptions comes under checked Exceptions.

* Throw keyword :-

→ throw keyword is mainly used to throw an exception explicitly.

→ we can specify (create) exception object which is to be thrown.

→ ~~we can say~~ by using throw keyword we can also throw the ~~is used to create~~ userdefined exception.

→ we can throw either checked or unchecked exception using throw keyword.

→ we can also define our own set of condition and throw ~~keep~~ exception explicitly.

→ Syntax-

throw new exception_class("error massage");

- Ex→ throw new IOException("Sorry Device not connected");

→

→ Throwing Unchecked Exception using throw keyword –

```
public class Test1 {
    public static void validate(int age) {
        if(age<18) {
            throw new ArithmeticException("Person is no eligible");
        }
        else {
            sop("Person is eligible");
        }
    }
}
```

→ ~~Creating~~ throwing User defined exception ~~using~~ using throw keyword –

```
class UserDefinedException extends Exception {
    public UserDefinedException(String str) {
        super(str);
    }
}

public class Test2 {
    psvm(String[] args) {
        try {
            throw new UserDefinedException("this is user defined Exception");
        }
        catch(Exception e) {
            ~~sop("Exception:"+ e);~~ e.printStackTrace();
            sop("Exception:"+e.getMessage);
        }
    }
}
```

* Throws keyword :-

→ the Java throws keyword is used to declare an exception.
→ Exception handeling is mainly used to handle the checked exception.
*→ we can declare only checked we can't declare errors, and unchecked exceptions.
→ throws keyword provides information to the caller of the method about exception.
→ throws keyword is used with method signature.
→ If we are calling a method that declares an exception, we must either caught or declare the exception:
there are two case:
case1 : we have handle the exception using try/catch.
case2: specified throws keyword with the method.
→ in case 1 the code will be executed fine whether exception occurs during the program or not.
→ in case 2 the code will be executed fine when exception does not occur but when exception occur then it will be thrown exception at runtime.

→ Case 1:

```
import java.io.*;
class Test {
    void method() throws IOException {
        throw new IOException("device Error");
    }
    psvm (String args[]) {
        try {
            Test t = new Test();
            t.method();
        } catch (Exception e) {
            sop("Exception:" + e);
        }
    }
}
```

→ Case 2:

```
import java.io.*;
class Test {
    void method() throws IOException {
        sop("Device operation completed");
    }
    psvm (String [] args) throws IOException {
        Test t = new Test();
        t.method();
        sop(" Bye ..... ");
    }
}
```

* Difference Between throw and throws:

throw	throws
① throw is mainly used to throw an exception explicitly.	① throws is mainly used to declare an exception.
② checked Exceptions are not propogated using throw keyword.	② checked Exceptions are propogated using throws keyword.
③ throw is used inside the method.	③ throws is used with method signature.
④ throw keyword is followed by instance of an Exception class.	④ throws keyword is followed by Exception class name.
⑤ we can throw only one Instance of an Exception class at a time.	⑤ we can declare multiple Exception classes at a time with the help of throws keyword.

⑥ throw Example:-

```
throw new ArithmeticException("Divided by Zero");
```

throws Example-

```
void show() throws IOException, ServletException { };
```

* Difference Between final, finally and finalize :-

① final is an access modifier but
finally is the block in Exception Handeling but
finalize is the method of object class.

② final modifier is used to create constant in java but
finally block is used to catch the Exception but
finalize method cut the all connections of a dereferenced object.

* Custom Exception :- (User Defined Exception).

```
class MyCustomException extends Exception {
        public MyCustomException (String str){
                super(str);
        }
}
public class Demo{
        psvm(String args[]){
                try{
                        throw new MyCustomException("Exception Arised");
                }
                catch(MyCustomException e){
                        SOP("Catch the Exception");
                        SOP(e.getMassage());
                }
        }
}
```

O/P → Catch the Exception
Exception Arised.

* try with Resource :-

try with resource is a new feature of java 9.

```
import java.io.FileOutputStreme;
class Demo{
    psvm(String args[]){
        try(FileOutputStreme f = new FileOutputStreme("/ambu/Document/tem.txt")){
            String msg = "Hello";
            byte byteArray[] = msg.getBytes();
            f.write(byteArray);
        }
        catch(Exception e){
            sop(e);
        }
    }
}
```

try with resource

Multithreading in Java :

* **Multitasking :-** multitasking is the ability of a computer to perform more then one task at the same time (concurrently).

→ this multitasking can be classified into two forms -
(1) Process Based Multitasking (Multiprocessing)
(2) Thread Based Multitasking (Multithreading)

* **Process :-** A process or a running process is a set of instruction (Program) concurrently run by a computer process.

→ Program under execution is process.
→ Ex→ MS office, VLC, etc.

* **Thread :-** Thread is a light weight subprocess which is considered as a smallest unit of process.

→ a process have multiple thread running concurrently and are destroyed as the task end or program ends.

* Process Based Multitasking :-

→ Process Based Multitasking allows multiple program or process to run concurrently at same time.
→ Process shares different memory area.
→ Process to Process communication is more expensive.
→ Process have lower data rate multitasking.
→ Process is a heavy weight. (heavely use of System resources).
→ A process can contain multiple thread.
→ Process are depends on each other. (50% true)
↳ as process are also ~~dependent~~ independent but threads are the part of process and thread is considered as dependent.

* Thread Based Multitasking (Multithreading) :-

→ Thread Based Multitasking allows multiple part (thread) of the same program (process) to run concurrently.
→ thread shares same memory area.
→ thread to thread ~~connection~~ communication is less expensive.
→ thread is a light weight subprocess.
→ multiple threads belongs to a same process.
→ threads are independent ~~of~~ each other. (50% true).

Note:-
→ For every program JVM create a "main" thread which is responsible for searching a main method in your program and if found initialise the class.
→ In other