S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
	Java folder			
	↳			
		Javac A.java ↵		
		Java A ↵		
		(or)		
	Java folder			

JAVA

Programming Language:

is a language which is used to write some instructions in order to get some work done by the s/m.

Pgm lang's generally classified into 2 types

1. Low level pgming lang.
2. High level pgming lang.

1. Low Level:

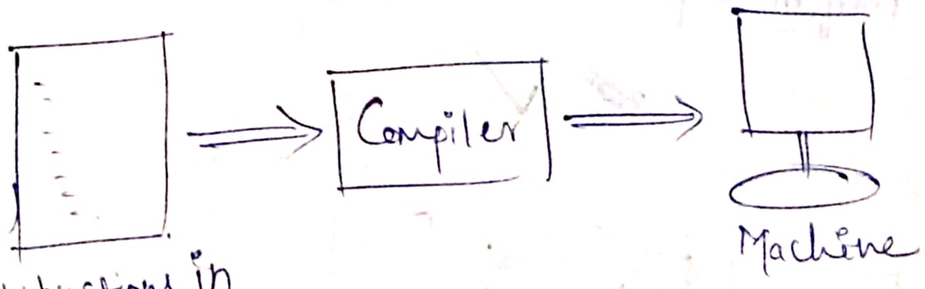
* A language which is easily understandable & executable by the machine is called low level pgming lang.

ex: Binary lang,

2. High Level

* A language which is easily readable understandable & instructable by the programmer is called high level pgming lang

ex: C, C++, java, python etc.



Instructions in High level pgming lang by programmer

* Compiler takes the instruction in high level pgming lang as input & converts into low level lang instructions
* The instn written in high level pgming lang are not directly understadable by machine therefore these instns must be compiled before executing.

Java :

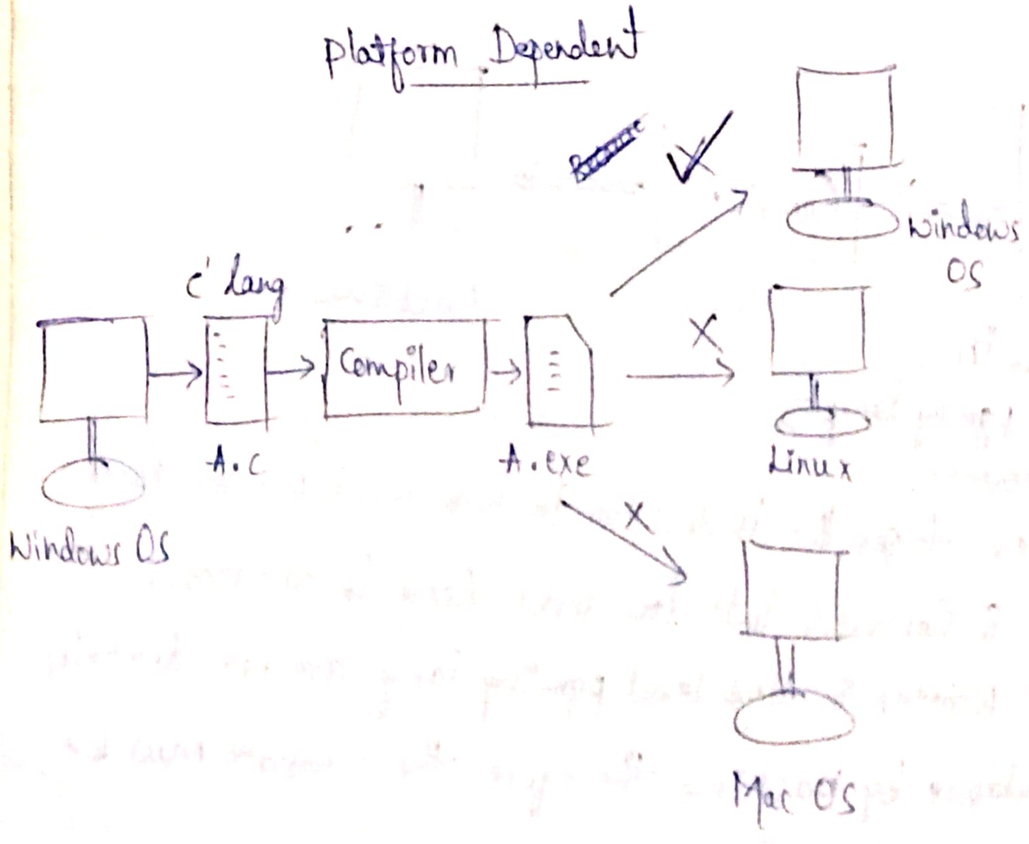
Java is a high Level pgming lang. developed by "James goesling" & his team under "Sun micro System".

* the first Version of java was released on 23rd Jan 1996

Features of Java

* High Level Programming Lang
* Simple
* Object Oriented Programming
* Comes With rich built in tools
* Platform Independent Lang.
* Secure
* Robust
* Multi threaded

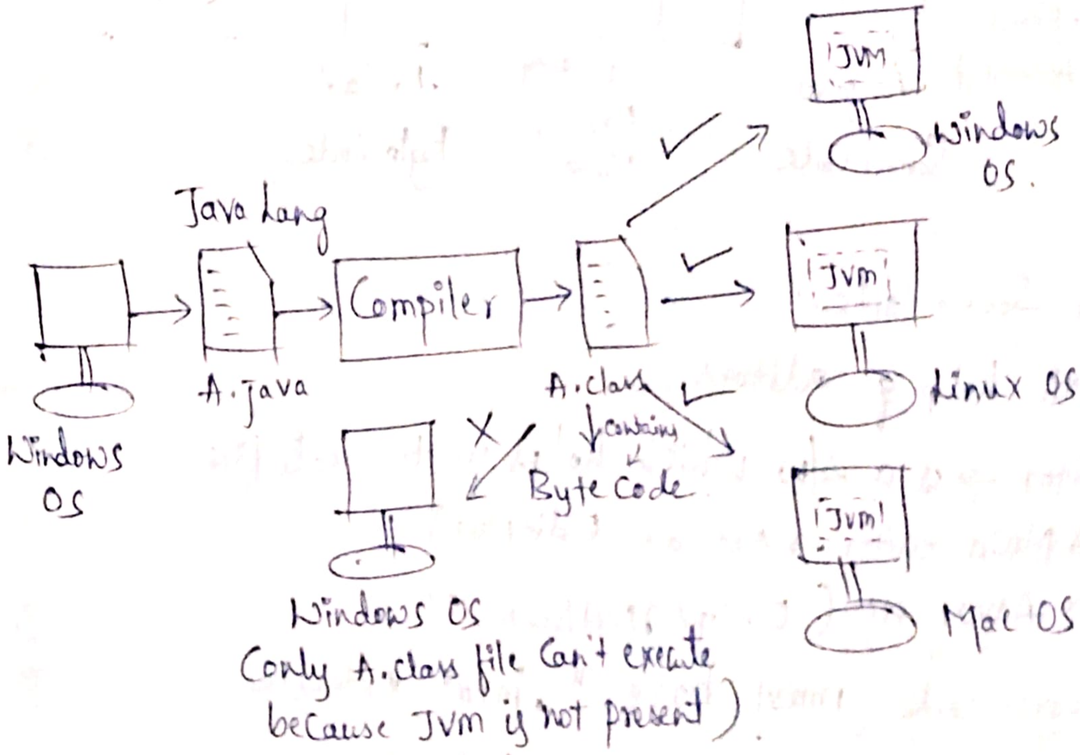
Platform Dependent Lang



Note:- In C lang the pgm Written (.c file) once Compiled generates an executable file (.exe). this .exe file Can be Can beexecuted Only in the platform which the pgmer is written, not in the other platforms as shown in example.:

∴ C is Called platform dependent lang.

* Platform Independent lang



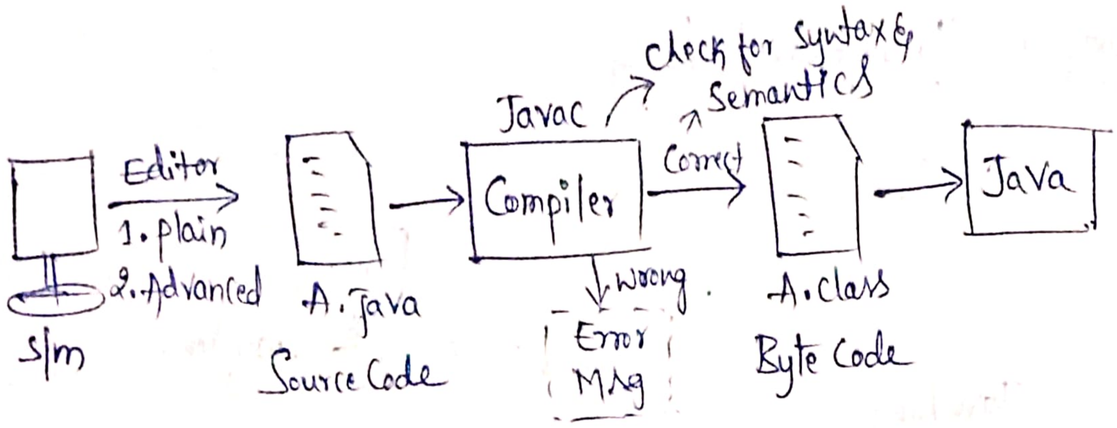
* In java, the pgmer Written (.java file) once Compiled will not be Converted to machine understabdable lang. Instead it gets Converted to an intermediate lang Called "Byte Code". This byte Code generated Can be executed in any platform which is having JVM installed in it.

* We Can't execute the byte code in a s/m which is not having JVM installed in it & there are diff Versions of JVM available for diff machines in the market.

∴ Java is Called "platform Independent" but JVM dependent lang.

Steps to Create, Compile, Execute Java Program

1. Create Source Code
2. Compile the Source Code & generate byte Code
3. Execute the byte Code



1. Create Source Code:

* With help of editors.
* Editor → is a s/w which helps us to Create files.
  - plain editor → (Notepad, Editplus)
  - Advanced (Eclipse, NetBeans, etc)
* Source code must have ".java" extension.

2. Compile Source Code & generate byte Code

* Compile Source Code (.java) file with help of Compiler.
* "Javac" is the cmd used for Compilation.
* Compiler Checks for Syntax & Semantics.
  if everything Correct → generate byte code
  wrong → Error Msg
* Byte Code will be saved in a class file with ".class" file extension.

3. Execution of Byte Code

* Byte code (or) .class file executed with help of java Command

Structure of Simple Java Program

```
Class Class_name
{
    Public Static Void main (String[] args)
    {
        statement/Instructions;
    }
}
```

Class

Main Method

Hello.Java

ex:

```
Class Hello
{
    public static Void main (String[] args)
    {
        System.out.println(" Hello World");
    }
}
```

Command Prompt

Compile :- Javac file_nam.Java
Javac Hello.Java

To change drive ⇒ drive_name : ↵
To enter folder ⇒ cd folder_name ↵

Execute: Java class_name | filename
Java Hello

Create new folder ⇒ mkdir fol_name ↵
(or)
md folder_name ↵

Note: Can We Create Class Without main method?

⇒ Yes, A Class Without main method Can be Compiled but it Cannot be executed.

Tokens of Java :-

* The smallest individual Components of a java pgm are called as tokens
  * Keywords
  * Identifiers
  * Literals
  * Seperators.

* Keywords

* Pre-defined words in java
* Reserved words
* Compile aware words.

Note: Keywords must be written in lower case

ex: class, public, int, static, void etc.

* There are 51 Keywords in java, Currently 49 are using.

Identifiers :

* Identifiers are the names given to elements of java pgm.

ex: Classname, methodname, Variable name etc.

Rules :

1. Keywords Cannot be used as a identifiers
2. Identifier should not start with a numeric Value.
3. Identifier Cannot have Space
4. Only 2 special Character Can be used as a identifier i.e $ -

Literals :-

* The type of values that can be used in a java pgm.

4 types

(i) Number
(ii) char
(iii) String
(iv) Boolean.

(i) Number :

(a) Integer ⇒ 10, 20, 3000

(b) Floating Point no ⇒ 0.4, 10.2, 4.5

(ii) Character :

* char must be enclosed by ' ' ⇒ 'a', 'b',
* length of char must be 1 ⇒ 'a', 'b', '2', 'ab' ✗ '12' ✗
* Case Sensitive ⇒ 'a' == 'A' ✗, 97 != 65 ⇒ ASCII values are diff

(iii) String :

* String must be enclosed with " " ⇒ "Hello", "A", "bye"
* length can be anything ⇒ "A", "2", "12345", "abcd"
* Case Sensitive ⇒ "a" != "A".

(iv) Boolean :

* There are 2 boolean values

(a) True
(b) False

Printing Statements

System.out.println();

* System.out.println ~~("hello")~~ Statement prints the given value & takes the cursor the next line

ex: S.O.println("Hello");
Hello
_ ← cursor

ex: S.O.println("Hello");
S.O.println("bye");
Hello
bye
_ ← cursor

* System.out.println Statement can be used without passing any value. (It takes the cursor to new line)

ex: S.O.pln( );
_
_ ← cursor

System.out.print();

* System.out.print Statement prints the given value & it will not take the cursor to next line.

ex: System.out.print("Hello");
Hello_ ← cursor

ex: S.O.print("Hello");
S.O.print("bye");
Hellobye_ ← cursor

* We cannot use S.O.print Statement without passing value, it generates compile time error.

ex: S.O.print( );
⇒ Compile time error.

Note: For printing statements we can pass any type of value

ex:
```
class G
{
    public static void main (String[]args)
    {
        S.O.pln(12); // Number
        S.O.pln('a'); // char
        S.O.pln("abcd"); // String
        S.O.pln(True); // Boolean
    }
}
```

ex ~~Kiric~~

```
class A
{
    public static void main (String[] args)
    {
        System.out.print (1);
        System.out.println(2);
        System.out.print ('A');
        System.out.print ('B');
        System.out.println('C');
        System.out.println (3);
    }
}
```

Note: For printing statement we can pass expression first that expression will be solved & result of the expression will be printe

ex:

```
class A
{
    p.s.v. main(String[] args)
    {
        s.o.pln (1+2);
        s.o.pln("1+2");
        s.o.pln("1"+"2");
    }
}
```

o/p
3
1+2
12

Plus Operator (+)

* has 2 behaviours

(i) Addition

(ii) Concatination

```
Class A
{
    public Static Void main(String[] args)
    {
        S.o.pln (10+20); // Num+Num // 30 // Num // Addition
        S.o.pln (10+'a'); // Num+char // 107 // Num // Add
        S.o.pln ('a'+'b'); // char+Char // 195 // Num // Add
        S.o.pln (10+"abcd"); // Num+String // 10abcd // String // Concat
        S.o.pln ('a'+"aa"); // Char+String // aaa // String // Concat.
        S.o.pln ("hello"+"hi"); // String+String // hellohi // String // Concat
    }
}
```

Note:- We Can't perform addition with boolean. but a boolean Value Can be Concatinated with a String.

ex:

```
Class A
{
    P.S.V. main(String[] args).
    {
        S.o.pln (10+true); // Num+boolean // Compile Time Error
        S.o.pln ('a'+false); // Num+boolean // CTE
        S.o.pln (true+false); // boolean+boolean // CTE
        S.o.pln (true+"Love"); // boolean+String // trueLove // String // Concat
    }
}
```

ex

```
class A
{
    P.S.V.m (String[] args)
    {
        S.o.pln(12+'a'+"hello"+100); → 109 hello 100
        S.o.pln('A'+100+"hi"+false); → 75 hi false
        S.o.pln('a'+"hello"+10+true); → a hello 10 true
        S.o.pln(false+10+"hello"+'a'); CTE
    }
}
```

o/p

109 hello 100. a hello 10 true

75 hi false ~~false 10 hello~~

Storing a Value:

Syntax: Datatype Variable = Value;

* We Can store a Value with the help of datatype, Variable & assignment operator.

Data Type:

* Datatype Specifies type of Value to be Stored in the memory block.
* Datatype decides size of the memory block.

2 types

(i) Primitive Datatype
(ii) Non-Primitive Datatype.

Primitive Datatype:

* the datatype which are used to Store Primitive Values Such as number, char (or) boolean are known as primitive data types.
* There are 8 primitive datatypes in java.

Type of Value	Datatype (8 datatypes)	Size
(1) Number		
(a) Integer →	Byte	1 byte
	Short	2 byte
	int	4 byte
	long	8 byte
(b) Floating →	float	4 byte
	Double	8 byte
(2) Character →	Char.	2 byte
(3) Boolean →	Boolean.	1 bit

Variable :

* Variable is name given to memory block which Contains a Value

2 types

(i) Local Variables

(ii) Global Variables.
- Static
- non-Static

(i) Local Variable :

* A Variable declared a inside a method (or) block is Called local Variable.

* Where as Variable declared directly inside the ~~the~~ Class is Called "global Variable".

ex:
```
class A
{
    int a; ← global
    {
      int c; ← local.
    }
    P.S.V.m(Args){
    {
      int b; ← local Variable //We Can access 'a' but
    }.                          We Can't access 'c'
}
```

* Local Variables Can be used Only inside the method (or) block in which they are declared. Where as global Variables Can be used through out the class.

* Local Variables must be initialize before using otherwise Compiler generates Compile Time error.

ex
```
class A
{
    P.S.V.m(String[]args)
    {
      int a;
      S.O.Pln(a); // CTE
    }
}
```

* Global Variables Can be used without initialization as they take default Values.

Initializing a Variable :

* We Can initialize (or) assign Value to the Variable With the help of assignment (=) operator.
* Assignment operator is used to assign the Value present on RHS to the Variable present on LHS.

ex:
```
class A
{
    p.s.v.m (String[] args)
    {
        int a;
        a = 28; //initialize o/p
        S.o.pln (a); // 28
        a = 50; //Re initializing
        S.o.pln (a); // 50
    }
}
```

Note :- inside a method or block we Can have more than one Variable but those Variable must have diff names. Other wise Compiler generate Compile time error

ex:
```
class A
{
    p.s.v.m (S[] args)
    {
        int a=10;
        int a=20; //CTE
        S.o.pln(a);
    }
}
```

```
class A
{
    p.s.v.m (String[] args).
    {
        int a=10;
        int b=20;
        S.o.pln (a); //20
        S.o.pln (b); //20
    }
}
```

Declaring & Initilizing Multiple Variables

* We Can declare & initialize multiple Variables using following Syntax

datatype V_1, V_2, V_3 - - - Variable n;
Variable 1 = Value 1;
Variable 2 = Value 2;
⋮

(or)

datatype Variable 1 = Value 1, Variable 2 = Value 2; ...

ex

```
class A
{
    p.s.v.m (String[] args)
    {
        int a=10;
        int b=20;
        char x,y,z;
        x='a';
        y='b';
        z='c';
        double e=10.2, f=12.2, g=14.5;
        S.o.pln (y); // b
        S.o.pln (b); //20
        S.o.pln (f); //12.2
    }
}
```

Operator:

Operators are the symbols used to perform operations.

* The values that are used for operations are "Operands".

ex. 10 + 20 → operator.
(10 ↓ operand, 20 ↓ operand)

* Based on the no of operands they take operators can be classified into 3 types

(i) Unary Operator:

The operator which take only one value (or) one operand are considered as

ex: inc operator, dec operator, cast operator.

(ii) Binary Operators:

The operators which takes two operands are considered as binary

ex: Arithmatic Operators, Relational Operators, Assignment operator, logical

(iii) Ternary Operator:

* the operators which takes three operands are considered as Ternary

ex: Conditional Operators.

Arithmatic Operators

* are binary operators +, -, *, /, %

ex:
```
class A
{
    p.s.v m(String[] args)
    {
        int a=10;
        int b=2;
        S.o.pln(a+b);
        S.o.pln(a-b);
        S.o.pln(a*b);
        S.o.pln(a/b);
        S.o.pln(a%b); // 0 ← Reminder
        S.o.pln(13%10); //3
        S.o.pln(3%10); //3
    }
}
```

ex:
```
class A
{
    p.s.v.m(String[] args)
    {
        int a=10, b=20;
        int Sum;
        Sum = a+b;
        S.o.pln("Sum is " + Sum);
        S.o.pln("Sum of " + a + " and " + b + " is " + Sum);
    }
}
```

ex:
```
class A
{
    p.s.v.m(String[] args)
    {
        int a = 26; b = a/10;
        System.out.Println("Last digit of " + a + " is " + b);
```

Increment Operator:

* Inc operator is used to increment the value of the variable by 1.
* Inc operator is denoted by ++
* Inc operator is unary operator.
* 2 types in inc operator.
  1. post inc
  2. pre inc

1. Post Increment Operator:

* If the increment operator is suffix to the variable then it acts as post inc. operator.

ex: Variable ++;
    a++;

Note: * Post inc operator first uses the value of the variable then it inc the value of variable by 1.

ex:
```
class A
{
    p.s.v.m(String[] args)
    {
        int a = 10;
        S.o.pln(a); //10
        S.o.pln(a++); //10
        S.o.pln(a); //11
        S.o.pln(a); //11
        S.o.pln(++a); //12
        S.o.pln(a); //12
    }
}
```

Post inc

Pre inc

ex: Class A

```
{
    p.s.v.m (String[] args)
    {
        int a=10;
        int b=20;
        s.o.pln (a++); //10
        s.o.pln (b++); //20
        s.o.pln (a++); 11
        s.o.pln (b++); 21
        s.o.pln (a); 12
        s.o.pln (b); 22
    }
}
```

② Pre Increment Operator:

* If the inc operator prefix to the Variable then it acts as pre inc operator.

ex: ++ Variable;

Note: pre inc operator first inc the value of Variable by 1 then it uses the incremented Value.

ex: Class A

```
{
    p.s.v.m (String[] args)
    {
        s.o int a=10;          a=10
                               b=11
        int b=++a; ⇒ first inc 'a', then assign a value to b.
        s.o.pln (++b); 12
        s.o.pln (++a); 12
        s.o.pln (++a); 13
        s.o.pln (++b); 13
        s.o.pln (a); 13
        s.o.pln (b); 13.
    }
}
```

ex:
```
class A
{
    p.s.v.m (String[] args)
    {
        int a = 5;
        int b = 6;
        S.o.pln (a++); //5
        int c = b++;
        S.o.pln (++c); //7
        S.o.pln (a++); //6
        S.o.pln (++b); //8
        S.o.pln (b++); //8
        S.o.pln (a); // 7
        S.o.pln (b); // 9
        S.o.pln (c); // 7
    }
}
```

Decrement Operator:

* Dec Operator is used to dec. value of the Variable by 1.
* It is denoted by --.
* It is an unary operator.
* There are 2 types of dec. operator.

1. Post decrement
2. Pre decrement

1. Post decrement Operator:

* If the dec. operator's suffix to the Variable then it acts as post dec operator.

Note: post decrement operator first uses the value of the Variable then it decrements value of Variable by 1.

ex:
```
class A
{
    p.s.v.m (String[] args)
    {
        int a=5;
        S.o.pln (a--); //5
    }   S.o.pln (a); // 4
}
```

② Pre decrement Operator:

* If the dec. operator is prefix to the Variable then it acts as pre dec operator.

*Note: pre dec. operator first dec value of the Variable by 1 then it uses the dec. Value.

ex: Class A
{ p.s.v.m (String [] args)
{ int a=5;
s.o.pln(--a); //4
s.o.pln (a); //4
} }

ex: Class A
{ p.s.v.m (string [] args)
{ int a=5;
int b=10;
s.o.pln (a--); //5
s.o.pln (--b); //9
int c=a--; ← first assign 'a's value to c, then dec.
s.o.pln (c--); //5
s.o.pln (--a); //2
b--;
s.o.pln (a); //2
s.o.pln (b); //8
s.o.pln (c); //3

ex: class A ***

```
{
  p.s.v.m (String[] args)
  {
    int a = 11;
    int b = --a; ← first dec var 'a' assign same value 'b'
    S.o.pln (b++); // 10
    S.o.pln (a--); // 10
    int c = ++b; ← 1st inc b then same value assign to c
    S.o.pln (--c); // 10
    S.o.pln (b--); // 12
    S.o.pln (a++); // 9
    ++a;
    S.o.pln (c+1); // 12
    S.o.pln (a); // 11
    S.o.pln (b); // 11
    S.o.pln (c); // 10
  }
}
```

a = ~~10~~ ~~9~~ ~~10~~ 11
b = ~~10~~ ~~12~~ 11
c = ~~11~~ 10

Relational Operator:

* Relational operators are binary operators & gives the output in boolean format (true, false).

ex:

==	10 == 10	→ true
!=	20 != 10	→ true
>	30 > 20	→ true
<	40 < 30	→ false.
>=	10 >= 5	→ true 10 >= 20 → false.
<=	11 <= 12	→ true.

ex: class B

```
{
  p.s.v.m (S args)
  {
    int a = 10; int b = 11;
    S.o.pln (a==b); // false
    S.o.pln (a!=b); // True
    S.o.pln (a++>b); // f
    S.o.pln (--a>b++); // ~~fa~~
    S.o.pln (++a>=--b); // True
    S.o.pln (b++<a); // f
```

a = ~~11~~ 10 ~~10~~ 11 ~~9~~ 10
b = 11 ~~11~~ 12 11

S.o.pln (a); // 11
S.o.pln (b); // 12.

Assignment Operator

* is a binary operator & it is used to assign the value present on RHS to the variable present in LHS.
* Assignment operator executed from Right to left.

ex: class A
{ p.s.v.m (string[] args)
{ int a, b;
a = b = 10;
s.o.pln(a); // 10
s.o.pln(b); // 10

int b = 10;
b* = b+2;
b* = 10+2
b = b*12
b = 120

Assignment Operators

=	int a = 20
+=	a += 2;
-=	a = a+2
*=	a = 22
/=	∴ a = 22
%=	

BODMAS

a = 10;
b = 2;
c = 1;
d = 0;
d += c * = b += a/2;
s.o.pln (d); // 7

d += c * = b += a/2;
b += a/2;
b += 10/2;
b += 5
b = b+5
b = 7

c * = b
c = c*b
c = 1*7
c = 7

d += c
d = d+c
d = 0+7
d = 7

Logical Operators:

AND (&&)

* AND operator is a binary operator & it is denoted by &&

Syntax: Exp1 && Exp2

* In above syntax expression 1 & exp2 must written a boolean value

Exp1	Exp2	Result
F	T	F
F	F	F
T	F	F
T	T	T

* If exp1 is false, exp2 will not be checked.

eg Class A

```
{
  p.s.v.m(s.a)
  {
    int a=10, b=11;
    s.o.pln(a==10 && b==11);//true
    s.o.pln(a==10 && b<a);// false
    s.o.pln(a++<b && b++>=a);//True
    s.o.pln(--a>++b && b--<=a++);// F
    s.o.pln(a<=b++ && b>a--);// T
    s.o.pln(a);//9
    s.o.pln(b);//14
  }
}
```

a = 11
b = 12

② OR (||)

* OR operator is a binary Operator.

Syntax: exp1 || exp2;

* In above Syntax exp1 & exp2 must written a boolean Value.

Exp1	Exp2	Result
F	F	F
F	T	T
T	F	T
T	T	T

* If the exp1 is "True", exp2 will not be checked & written as "True"

eg Class A

```
{
  p.s.w.m(S[] a)
  {
    int a=5, b=4;
    s.o.pln(a>b || b<=4);//true
    s.o.pln(a<b || b==4);//true
    s.o.pln(a++>b-- || b++<a++);// True
    s.o.pln(a<--b || ++b<++a);// True
    s.o.pln(--b<=a-- || ++a<=b);// True
    s.o.pln(a);//6
    s.o.pln(b);//2
  }
}
```

a = 8 6 7 6
b = 4 3 2 3 2

Conditional Operators

* is ternary Operator.
* Syntax: Expression ? Statement 1 : Statement 2

  True → Statement 1

  False → Statement 2

* In above Syntax expression must written as boolean Value. if it returns true Statement 1 will be executed otherwise Statement 2 will be executed.

ex:
```
class A
{
    p.s.v m(S[] args)
    {
        S.o.pln(true?"hello":"bye"); //hello
        S.o.pln(false?"hello":"bye"); // bye
    }
}
```

* Write a pgm to print largest 2 no's using Condition Operator.

```
class A
{
    p.s.v m(String[] args)
    {
        int a=10, b=5;
        S.o.pln(a>b?"A is greater"+a:"b is greater"+b);
        int largest = a>b?a:b;
        S.o.pln("Largest Number is "+largest); // 10
    }
}
```

* Write a pgm to Check whether a number is even or odd using Conditional Operator.

```
class A
{
    p.s.v.m(String[] args)
    {
        int a=100;
        S.o.pln(a%2==0?"Even":"odd");
    }
```

* Write a pgm to display smallest of 3 numbers using Conditional Operator.

```
Class A
{
    p.s.v.m (S[] args)
    {
        int a=10, b=2, c=12;
        int smallest1 = a<b ? a:b;
    int smallest2 = smallest1 < c ? smallest1 : c;
        s.o.p("smallest number is"+smallest2);
    }
}
```

* Write a pgm to check whether divisible by 3 as well as 5. If it divisible by both 3 & 5 print a msg saying divisible otherwise print not divisible.

```
Class A
{
    p.s.v.m (S[] args)
    {
        int num=15;
    s.o.pln (num%3==0 && num%5==0 ? "Divisible":"Not");
    }
}
```

* Smallest of 3 numbers.

```
int a=12, b=120, c=37;
s.o.pln (a<b && a<c ? a : (b<c ? b:c));
```

Decision Making Statement

* are the statements that are used to check a Condition before executing a set of instructions.

If
If else
if-else ladder
Nested decision making statement
Switch.

If :-

Syntax: if (Condition) ← single Statement
statement ;
(or)
if (Condition)
{
statement ;
}

ex: class A
{
p s v. m (String [] args)
{
s.o.pln (1); →
if (true) → True it will print 2
s.o.pln (2); →
if (false) → false so skip the s.o.p (3);
s.o.pln (3); →
if (false)
s.o.pln (3); →
s.o.pln (4);
if (true)
s.o.pln (5);
s.o.pln (6);
}
}

o/p
1
2
4
5
6.

ex:

```
class A
{
    p.s.v.m (String[] args)
    {
        s.o.pln ("Idli")
        if (true) ---> True means print the msg given {
        {
            s.o.pln ("poori");
            s.o.pln ("chapathi");
        }
        s.o.pln ("B");
        if (false)
        {
            s.o.pln ("C");
            s.o.pln ("D");
        }
        if (false); ---> means end of if
        s.o.pln ("G");
        s.o.pln ("H");
    }
```

o/p

Idli
poori
chapathi
B
G
H.

(*) Write a pgm to check whether the no. even using if statement.

```
class A
{
    p.s.v.m (String[] args)
    {
        int a = 50;
        if (a % 2 == 0)
        {
            s.o.pln ("Even Number")
        }
        ~~s.o.pln~~ if (a % 2 == 1)
        {
            s.o.pln ("odd no");
        }
    }
```

If-else :

Syntax:

```
if (Condition)
    Statement 1;
else
    Statement 2;
```

Single Statement ⟸ else

```
if (condition)
{
    Statement 1;
}
else
{
    Statement;
}
```

⇒ Multiple Statements

Ex:

```
class A
{
    p.s.v.m (s[] args)
    {
        s.o.pln (1);
        if(true)
        {
            s.o.pln(2);
            s.o.pln(3);
        }
        else
        {
            s.o.pln(4);
            s.o.pln(5);
        }
        s.o.pln(6);
        if(false)
        s.o.pln (7);
        else
        s.o.pln(8);
        s.o.pln (9);
    }
}
```

o/p

1
2
3
6
8
9

* Write a pgm to check whether last digit of the number is even or odd

```
Class A
{
    p.s.v.m (String[] args)
    int a=58;
    int b = a/10 ? a : c;
    if (a%2==0)
    {
        s.o.pln ("Even");
    }
    else
    {
        s.o.pln ("odd");
    }
}
}
```

```
int ld = a%10;
if(ld%2==0)
{
    even
}
else
{
    odd
}
```

If-else ladder :

Syntax:

```
if (condition 1)
{
  statements;
}
else if (Condition2)
{
  statements;
}
else if (Condition3)
{
  statements

else
{
  statements;
}
```

Note: * In if else ladder one Condition is satisfied remaining Conditions will not be checked.

* else block in if else ladder is not mandatory

(*) pgm to find largest of 3 no's using if else ladder.

```
class A
{
  p.s.v.m (S[] args)
  int a = 10, b=20, c=30;
  if (a > b && a>c)
  {
    s.o.pln("A is largest"); || s.o.pln(a+"is largest");
  }
  else if (b>c)
  {
    s.o.pln("B is largest")
  }
  else
  {
    s.o.pln("C is largest");
  }
}
```

* Pgram to print.

A → if the char is upper case alphabet
B → if the char is lowercase alphabet
C → if the char is digit
D → if the char is space
Z → for all remaining special char.

```
class A
{
  ~~int a=10;~~ p.s.v.m(String[] args)
  {
    char ch='B';
    if(ch>='A' && ch<='Z')
    {
      s.o.pln("A");
    }
    else if(ch>='a' && ch<='z')
    {
      s.o.pln("B");
    }
    else if(ch>='0' && ch<='9')
    {
      s.o.pln("C");
    }
    else if(ch==' ')
    {
      s.o.pln("D");
    }
    else
    {
      s.o.pln("Z");
    }
  }
}
```

* Write a pgm to print

Hi → if the no. is even & divisible by 5

Hello → if the no. is just even

bye → if the no. is odd & divisible by 3 as well as 5

goodbye → if ~~the~~ ~~name~~ none of the above conditions are satisfied

```
class A
{
    p.s.v.m (String[] args)
    {
        int a = 30;
        if (a%2==0 && a%5==0)
        {
            s.o.pln ("Hi");
        }
        else if (a%2==0)
        {
            s.o.pln ("Hello");
        }
        else if (a%3==0 && a%5==0)
        {
            s.o.pln ("Bye");
        }
        else
        {
            s.o.pln ("Good Bye");
        }
    }
}
```

(*) Take character & Write a pgm

(1) Is it alphabet

(a) Is it lower case?

(i) is it vowel? → Print "It is lower case vowel".

(ii) else print "It is lower case "Consonent".

(b) Is it upper case?

(i) is it vowel? print "It is a upper case vowel".

(ii) else print → "It is a lower case consonant.

Nested decision making statement

* Writing a decision making statement inside another decision making called nested decision making statement.

Ex: ② If it's digit? → print "It is a digit".
③ If it is a space? print → "It is a space.
④ print 'It is a special char' if non of the above condition is satisfied.

```
class A
{ p.s.v.m(String[] args)
  {
    char ch='a';
    if(ch>='A' && ch<='Z'||ch>='a' && ch<='z')
    {
      if(ch>='a' && ch<='z')
      {
        if(ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u')
        {
          s.o.pln("It is lower case Vowel");
        }
      else
        {
          s.o.pln("It is lowercase Consonant");
        }
    }
    else if(ch>='A' && ch<='Z')
    {
      if(ch=='A'||ch=='E'||ch=='I'||ch=='O'||ch=='U')
      {
        s.o.pln("It is upper case Vowel");
      }
      else
        {
          s.o.pln("It is upper case Consonant");
        }
    }
    else if(ch>='0' && ch<='9')
    {
      s.o.pln("It is a digit");
    }
    else if(ch==' ')
    {
      s.o.pln("It is a space");
    }
    else
    {
      s.o.pln("It is a special character");
    }
  }
}
```

Switch :

* Switch is a special decision making statement.
  It checks for "equality".

Note * for switch we can pass a Value/Variable/Expr

* Switch accepts any type of Value other than "float, double & boolean".

Syntax: switch (value)
{
Case 1 : Statement;
break;
Case 2 : Statement;
break;

[default : Statement]
}

ex:
```
class A
{
  p.s.v.m (String[] args)
  {
    int a = 10; switch(a)
    {
      Case 10 : S.o.pln ("Value is 10");
                break;
      Case 20 : S.o.pln ("Value is 20");
                break;
      default : S.o.pln ("Not Matching");
    }
  }
}
```

o/p: Value is 10.

Note: * The Value passed to Switch will be Compared with Case Values (Checking for equality), if there is any matching Case Value found then the statements for Case gets executed.

* If there is no matching Case Value, then default gets executed & using default in Switch is not mandatory

* Break is a Control transfer statement used to transfer the Control out of Switch block.

ex:
```
class A
{ p.s.v.m (a[]s)
  {
    int a = 10;
    switch (a)
    {
      Case 10 : S.o.pln ("Value is 10");
                break;
      Case 20 : S.o.pln ("Value is 20");
                break;
    } default : S.o.pln ("Not Match");
  }
}
```

o/p Value is 10

* Write a pgm to check whether a char is lower case vowel or not using switch

```
Class A
{
    p.s.v.m(String[] args)
    {
        char a = 'e';
        switch (a)
        {
            Case 'a' : S.o.pln("Vowel a");
                       break;
            Case 'e' : S.o.pln("Vowel e");
                       break;
            Case 'i' : S.o.pln("Vowel i");
                       break;
            Case 'o' : S.o.pln("Vowel o");
                       break;
            Case 'u' : S.o.pln("Vowel u");
                       break;
            default : S.o.pln("Not a Vowel");
        }
    }
}
```

o/p

Vowel e

* Write a pgm to check whether a number is even or odd using switch

```
class A
{
    p.s.v.m(String[] args)
    {
        int a = 27; int b = a % 2;
        Switch (b)
        {
            Case 0 : S.o.pln("Number is even");
                     break;

            default : S.o.pln("Numer is odd");
        }
    }
}
```

Looping Statement

* Looping statement are the statements that are used to execute an instn multiple times.

1. While
2. Do-while
3. For loop

1. While :-

Syntax:
```
while (Condition)
{
    Statement;
    Update;
}
```

* Write a pgm to print Hello 5 times in screen.

```
class A
{
  p.s.v m(S[] args)
  {
    int i=1;
    while (i<=5)
    {
      s.o.pln("Hello");
      i++;
    }
}}
```

O/P
Hello
Hello
Hello
Hello
Hello.

* Write a pgm to print numbers from 1 to 10

```
class A
{
  p.s.v.m(S[] args)
  {
    int i=1
    while (i<=10)
    {
      s.o.pln(i);
      i++;
    }
}}
```

* Write a pgm to print only even numbers in the range 1 to 10

```
Class A
{
  ps s. v. m (String[] args)
  {
    int i = 2;
    while (i<=10)
    {
      s.o.pln(i);
      i=i+2;
    }
  }
```

(or)

```
int i = 1;
while (i<=10)
{ if (i%2==0)
  {
    s.o.pln(i);
  }
  i++;
}
```

* Write pgm to print both even & odd numbers in the range 20 to 30 with suitable msg's.

```
class A
{
  p.s.v.m (String[] args)
  {
    int i = 20;
    while (i<=30)
    {
      int j = i%2;
      if(j==0)
      {
        s.o.pln(i+ "Is even no");
      }
      else
      {
        s.o.pln(i+" is odd no");
      }
      i++;
} } }
```

* Write a pgm to display Sum of all the numbers in range 1 to 5

```
Class A
{
    p.s.v m(String[] args)
    {
        int i=1;
        int Sum=0;
        while(i<=5)
        {
            Sum=Sum+i;
            i++;
        }
        S.o.pln("Sum is" + Sum);
    }
}
```

* Write a pgm to display Sum of odd no's in the rang 50 to 100.

```
Class A
{
    p.s.v.m(String[] args)
    {
        int i=50;
        int Sum=0;
        while(i<=100)
        {
            if(i%2==1)
            {
                Sum=Sum+i;
            }
            i++;
        }
        S.o.pln("Sum is"+Sum)
    }
}
```

* Write a pgm to Count no. of digits in a given number

```
Class A
{
    p.s.v.m(String[] args)
    {
        int i=123;
        int Count=0;
        while(i>0)
        {
            i=i/10;
            Count++;
        }
        S.o.pln("Total no. of digits" + Count);
    }
}
```

o/p
3

123%10 → 3 ← last digit
123/10 → 12 ← actually 12.3
both are integers

* Write a pgm to display all the digits of a number one by one starting from last digit

```
Class A
{
  int num = 123;
  int ~~count = 0~~; ld;
  while (num != 0)
  {
  ~~ld = num % 10;~~
    ~~S.o.p~~
    ld = num % 10;
    num = num / 10;
  }
}
```

```
int num = 123;
int ld;
while (num > 0).
{
  ld = num % 10;
  S.o.pln(ld);
  num = num / 10;
}
}
```

o/p
3
2
1

* WAP to print the sum of last digits

```
Class A
{
  int num1 = 256;
  int num2 = 172;
  int ld1, ld2; int Sum = 0;
  while (num1 > 0 && num2 > 0)
  {
    ld1 = num1 % 10;
    ld2 = num2 % 10;
    Sum = ld1 + ld2 + Sum;
    ~~S.o.pln (Sum)~~ num1 = num1 / 10;
                      num2 = num2 / 10;
  }
  S.o.pln (Sum);
}
}
```

* WAP to print sum of odd no's & sum of even no's seperately in the range 10 to 20.

```
Class Sumevenodd
{
  P.S.V.M (String[] args)
  {
    int i=10;
    int Sum1, Sum2=0;
    while (i<=20)
    {
      if (i%2==0)
      {
        Sum1 = Sum1+i;
      }
      else
      {
        Sum2 = Sum2+i;
      }
      i++;
    }
    S.O.pln ("Sum of even No's" +Sum1);
    S.O.pln ("Sum of odd No's" +Sum2);
  }
}
```

* WAP to display factorial of a number.

```
int num=4, fact=1
while (num>=1)
{
  fact = fact*num;
  num--;
}
S.O.pln(fact);
}
}
```

* WAP to find sum of all the digits in a given number.

```
Class A
{
  P.S.V.M()
  {
    int num=123;
    int Sum=0;
    int ld;
    while (num>0)
    {
      ld = num%10;
      Sum = Sum+ld;
      num = num/10;
    }
    S.O.pln("Sum of all digits" +Sum);
  }
}
```

WAP to print the Count of even digits in a given number.

```
class Sumeven
{ p.s.v.m(String[] args)
    int num=2574268;
    int Count=0;
    int ld;
    while (num>0)
    {
        ld=num%10;
        if(ld%2==0)
        {
            Count++;
        }
    } num=num/10; <- Update Num (Remove last digit)
    s.o.pln (Count);
} }
```

Control Transfer Statements

* are the statements that are used to transfer the Control from one statement to another..

* Break
* Continue

(1) Break :-

* is a Control Transfer Statement that is used to transfer the Control out of the switch or loop.

ex Syntax:

```
while (Condition)
{
    Statements;
    break;
}
Statements;
```

Note: * Break must be used as last statement of the block,

* If there is any statement after break in the ~~block~~ same block it generates Compile time error.

ex:

```
while(i%3==0)
{
    break;
    i++; // CTE
}
```

ex

```
class A
{ p.s.v.m(String[]args)
  { int i=1;
  while(i<=5)
  { s.o.pln (i);
    i++;
    break;
  }
  s.o.pln (i);
}
```

o/p

1

2

② Continue :

* Continue is a Control transfer Statement which transfers the Control back initial group Condition

Syntax
```
while (Condition)
{
    Statement 1;
    Continue;
}
```

Note:

* Continue must be the last Statement of the block if we write any Statement after Continue in the same block it generates Compile time error.

ex:
```
while (i<=4)
{
    if (i==3)
    {
        Continue; //CTE
        i++;
    }
}
```

ex:
```
class A
{
    int i=1;
    while(i<=4)
    {
        s.o.pln("*");
        if(i==3)
        {
            i++;
            Continue;
        }
        s.o.pln(i);
        i++;
    }
```

o/p
```
*
1*
2*
*
4
```

* Print 11 to 20 using while & Continue Skip 14 & 18

```
class A
{
    int i=11;
    while (i<=20)
    {
        ~~s.o.pln(i)~~
        if(i==14 || i==18)
        {
            i++;
            Continue;
        }
        s.o.pln(i);
        i++;
    }
}
```

→ Continue is a Control transfer statement which transfers the Control back initial group Condition

Syntax
```
while (Condition)
{
    Statement 1;
    Continue;
}
```

Note:

* Continue must be the last statement of the block if we write any statement after Continue in the same block it generates Compile time error

ex
```
class A
{
    int i=1;
    while (i<=4)
    {
        s.o.pln(i*i);
        if(i==3)
        {
            i++;
            Continue;
        }
        s.o.pln(i);
        i++;
    }
}
```

o/p

```
1 1
2 4
3 9
4 16
*
1 *
2 *
*
4
```

ex
```
while (i<=4)
{
    if (i==3)
    {
        Continue; //CTE
        i++;
    }
}
```

* Print 11 to 20 using while & Continue skip 14 & 18

```
class A
{
    int i=11;
    while (i<=20)
    {
        ~~s.o.pln(i)~~
        if(i==14 || i==18)
        {
            i++;
            Continue;
        }
        s.o.pln(i);
        i++;
    }
}
```

*WAP to print no's from 10 to 20 in b/w if there is any number divisible by 3 as well as 5 stop it.

```
Class A
{
  p.s.v.m(String[] args)
  {
    int i=10;
    While (i<=20)
    {
      if(i%3==0 && i%5==0)
      {
        ~~S.o.pln(i);~~
        break;
      }
      ~~i++;~~
    } S.o.pln(i);
    i++;
  }
}
```

*WAP to print only odd no's from 100 to 50 in b/w if there is any odd no. divisible by 3 as well as 9 stop it.

```
Class Onlyodd
{
  p.s.v.m(String[] args)
  {
    int num=100;
    While (num>=50)
    {
      if(num%2==1)
      {
        if(num%3==0 && num%9==0)
        {
          S.o.pln("The no"+num+" is divided
                    by 3 and 9");
        }  break;
      } end of if
      S.o.pln(num);
      num--;
    }// end of While
  }
}
```

② Continue :

* Continue is a Control transfer Statement which transfers the Control back initial group Condition

Syntax:

```
while (Condition)
{
    Statement 1;
    Continue;
}
```

Note:

* Continue must be the last Statement of the block if we write any Statement after Continue in the Same block it generates Compile time error.

ex:

```
while (i<=4)
{
    if (i==3)
    {
        Continue; //CTE
    }
    i++;
}
```

ex:

```
class A
{
    int i=1;
    while (i<=4)
    {
        S.o.pln("*");
        if (i==3)
        {
            i++;
            Continue;
        }
        S.o.pln(i);
        i++;
    }
```

o/p

```
*
1*
2*
*
4
```

* Print 11 to 20 using while & Continue Skip 14 & 18

```
class A
{
    int i=11;
    while (i<=20)
    {
        ~~S.o.pln(i)~~
        if (i==14 || i==18)
        {
            i++;
            Continue;
        }
        S.o.pln(i);
        i++;
    }
}
```

Break	Continue
* It transfers the Control out of the loop	* It takes the Control back initial loop Condition.
* It used to stop the execution	* It used to skip the execution
* It Can be used inside both switch, loop	* It Can be used only inside the loop.

Do-While:

Syntax:

```
do
{
    statement;
    update;
}
while (Condition);
```

Note: In do-while loop the statements present inside loop block will be executed first & then the loop Condition will be checked.

* In do-while loop while Condition must end with semicolon (;)

ex WAP print 1 to 10 no's using do-while.

```
Class A
{   p s. v m(args[] String)
    {
        int = 1;
        do
        {
            s.o.pln(i);
            i++;
        }
        while (i<=10);
    }
}
```

* In do-while loop even if the loop Condition is false for the 1st iteration the statements will executed atleast once.

For Loop :-

Syntax
```
for(Sec 1; Sec 2; Sec 3)
{
    Statement 1;
}
```

```
for (Initialization; Condition; Update)
{
    Statement;
}
```

1 2 3 4 5

* WAP to print no's from 1 to 10 using for loop.

```
class A
{
    p.s.v.m ( )
    {
    for(i=1; i<=10; i++)
    {
        S.O.pln (i);
    }
    }
}
```

* WAP to print a to h using for loop.

```
Class A
{
    p.s.v.m()
    {
        for(Char c='a'; c<='h'; c++)
        {
            S.o.pln(c);
        }
    }
}
```

* WAP to print 1st 17 char's/letters from alphabets.

```
class A
{
    p.s.v.m()
    {
        Char ch='a';
        for(int i=1; i<=17; i++)
        {
            S.o.pln(ch);
            ch++;
        }
    }
```

To take integer Number as user input:-

Step 1:- import Scanner class from java.util package.

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

Step 2: Create an object of Scanner class

```
Scanner s = new Scanner(System.in);
```

Step 3: Receive the number using next int method.

```
s.nextInt();
```

ex:

```
import java.util.Scanner;
Class A
{
    p.s.v.m(String[] args)
    {
        Scanner s = new Scanner(System.in);
        S.o.pln("Enter the number");
        int num = s.nextInt();
        S.o.pln("Entered Value is : " +num);
    }
}
```

* WAP for addition of 2 no's by taking user input.

```
import java.util.Scanner;
Class A
{
    p.s.v.m(String[] args)
    {
        Scanner s = new Scanner(System.in);
        S.o.pln("Enter 1st No : ");
        int a = s.nextInt();
        S.o.pln("Enter 2nd No : ");
        int b = s.nextInt();
    int sum = a+b;
    S.o.pln("The Sum of 2 no's : " + sum);
    }
}
```

Type Casting:

The process of Converting one type Value to another type is Called type Casting.

* Classified into 2 types

1. Primitive type Casting
   - (a) Widening
   - (b) Narrowing.
2. Non- Primitive type Casting
   - (a) Up Casting
   - (b) down Casting.

1. Primitive type Casting:-

* The process Converting one primitive data type to another data type is Called primitive type Casting.

* Classified into 2 types
  - (a) Widening
  - (b) Narrowing.

(a) Widening.

The process of Converting a narrow data type into wider data type is Called Widening.

char → int → long → float → double.

Narrow Data type

Wider data type.

* Widening is automatically done by the Compiler because there is no possiblity of data loss.

* Widening is also Called as autowidening.

ex:
```
class A
{ p s.v m (String[] args)
  {
    char ch='a';
    S.o.pln(ch);
    int x=ch; ← char to int i.e widening.
  } } S.o.pln(x);
```

ex: class A
{
p.s.v.m(String[] args)
{
int a=65;
double b=a; ← int to double i.e Widening
s.o.pln(b);
}
}

o/p
65.0

(b) Narrowing:-
The process of Converting a wider datatype into narrow datatype is called narrowing.

char ← int ← long ← float ← Double

* Narrowing is not automatically done by the Compiler because there is a possibility of data loss.
* It should be explicitly done by the programmer with help of Cast operator.

* Cast Operator:-
* Cast operator is a ~~Unary~~ Unary operator is used to perform narrowing

Syntax (datatype) Operand; → Values/variables/Expr

ex: class B
{
p.s.v.m(~~String~~[] args)
{
int a=98;
char ~~ch~~ (char)a; // int to char i.e Narrowing
s.o.pln(ch);
}
}

o/p
b. ← 98 is ASCII of b.

ex: class B
{
p.s.v.m(String[] args)
{
double a=65.25;
int b=(int)a; // double to int
s.o.pln(b); // 65
}
}

* A frog can jump the distance covered the frog has to cover the distance calculate the no of jumps required

jump is 1.8 meter now 30 meters from A to B

int/double
Result is double

```
class A
{
    P.S.V.M(String[] args)
    {
        double f = 1.8; int d = 30;
        ~~int a = (int) f;~~  ~~int c = d/f;~~  ~~int x = (d/f)~~
        ~~int b = (int) d;~~  ~~int x = (int) c;~~  int x = (int)(d/f);
        ~~int c = b/a;~~  ~~S.o.pln(x);~~  S.o.pln(x);
        ~~S.o.pln(c);~~
    }
}
```

Method :

Method is a block of instruction which is used perform some task

Syntax :-

```
Modifiers returntype methodname ([datatype Var1, datatype Var2 ])
{
    Statements/Instructions;
}
```

Header
method signature
formal argu

Method body/
Method implementation.

Method Header

* Includes.
  (i) Modifier
  (ii) returntype
  (iii) Methodname
  (iv) formal arguments.

Method Signature

* Combination of method name & formal argument is known as.

Method body / Method implementation

* The block of the method ~~with~~ which includes the instructions to be executed, is known as method body.

Return type:

* It specifies the type of value returned by the method.
* It can be Primitive datatype, Non-Primitive, Void. based on the type of value returned by the method.
* If method returns Primitive ~~datatype~~ value then return type must be suitable primitive datatype.
* If method returns non-primitive value then return type must be suitable non-primitive data type.
* If method doesn't return any value, then return type must be Void.

Void :- Void is a special return type which specifies no value.

Method_name:

* is a identifier therefore we must follow all the 4 rules of identifiers while writing methodname & also we should follow following Conventions.

↳ Conventions for methodname

* If methodname is a single word → all the letters should be lowercase
* ex: test(), print() etc...
* If it is multiword, from the second word onwards each new word should start with uppercase

ex: toPrint(), toPrintEvenNumbers(),

Formal Arguments

* The variable declared in the method header are known as
* these variables are local variables.

Note:- * A method gets executed only when it's called.

* Main method will be automatically called by JVM.

Method Call Statement :-

* is used to transfer the control from calling method to called method.

Syntax: Methodname (Actual Argument);

Actual Argument :-

* are the values passed to formal arguments of a method.

ex:
```
class A
{
    public static void test()  // Called method.
    {
        S.o.pln("from test()");
    }

    public static void main(String[] args)// Calling method.
    {
        S.o.pln("main() starts");
        test();// Call. Method Call Statement
        S.o.pln("main() ends");
    }
}
```

o/p

main() starts
From test()
main() ends.

Calling Method:

* A method which calls another method for execution is called calling.

Called Method:

* A method which is been called by another method is known.

Advantages of method

* No. of lines of code will be reduced.
* Code reusability.
* It makes the code more readable.

* While Calling a method with formal Argument

* Length of actual argument & formal arguments must be same
* Type of actual argument & formal argument must be same
  (or) We Can have an actual argument which Can be Widen to formal argument type.

ex:

```
Class B
{
  public Static Void add (int a, int b)
  {
    int sum = a+b;
    S.O.pln (sum);
  }
  public Static Void main(String[] args)
  {
    add (1,2); // 3
    add (10); // CTE
    add (10,20,30); // CTE
    add (1, 'a'); // 98 => Widening i.e automatically Converted
    add (1, 12.3); // CTE => Narrowing i.e not done automatically
  }
}
```

* WAP to design a method to check method no. even or odd.

```
Class A
{
  p.s.v.m (String[] args)
  {
    test (20);
  }

  public Static Void test (int a)
  {
    if (a%2 == 0)
    {
      S.O.pln ("Even number");
    }
    else
    {
      S.O.pln ("odd number");
    }
  }
```

* WAP to design a method to print numbers in the given range.

```
Class A
{
  p.s.v.m (String[] args)
  {
    test(20,50);
  }

  public static Void ~~main~~test (int a, int b)
  {
    ~~while (i<=50)~~                  for (int i=a; j<=b; i++)
    ~~{~~                                {
    ~~s.o.p~~                              s.o.pln (i);
    for (~~int i=a~~ int i; b<=50; i++)  }
    {
      ~~s.o.pln (a);~~
    }
  }
}
```

Return Statement:

* is used to transfer the Control from Called method to Calling method.
* whenever a method returns a value it's return type should be some data type other than Void.

ex:

```
Class A
{
  public static add (int a, int b)
  {
    int Sum = a+b;
    return Sum;
  }
  p.s.v.m (String[] args)
  {
    add (10,20); //return value not stored & not used // No output
    s.o.pln (add (10,20)); //return value just used not stored // 30
    int result = add (10,20); //return value stored & used //
    s.o.pln (result); //30.
  }
}
```

Note: Return Statement Can return only One value.

ex: return a; ✓
return a, b; ✗
return a+b; ✓
return 20, 30; ✗

ex: class A

```
{
  public static int test()
  {
    s.o.pln("from test()");
    return 10;
  }
  p.s.v.m(String[] args)
  {
    s.o.pln("from main()");
    int a = 50+test(); //a=60 ← 50+10
    s.o.pln(a);
  }
}
```

O/P

60

ex: class A

```
{
  public static void test()
  {
    s.o.pln("from test()");
  }
  p.s.m.v(String[] args)
  {
    s.o.pln(test()); //CTE
    int num = test(); //CTE
  }
}
```

→ Because test() is not returning any value.

* WAP to design a method which returns sum of even no's in given range

```
class A
{
  p.s.v.m(String[] args)
  {
    int result = even(1,10);
    s.o.pln(result);
  }
  public static void ~~main~~ even(int a, int b)
  {
    int sum=0;
    while (a<=b)
    {
      if (a%2==0)
      {
        sum=sum+a;
      }
      a++;
    }
  }
}
```

* design a method which returns Count of digits in a given no.

```
class B
{
  p.s.v.m(String[] args)
  {
    int result = test(1234);
    s.o.pln(result);
  }
  public static Void test(int a)
  {
    int Count=0;
    while(a>0)
    {
      a=a/10;
      Count++;
    }
    return Count;
  }
}
```

* design a method which returns Sum of Square of each digit of a given no

```
class C
{
  p.s.v.m(String[] args)
  {
    s.o.pln(test(123));
  }
  public static Void test(int num)
  {
    int Sum=0, ld, Sq;
    while(num>0)
    {
      ld = num%10;
      Sq = ld * ld;
      Sum = Sum + Sq;
      num = num/10;
    }
    return Sum;
  }
}
```

Note: * Return Statement Can return only one Value.

* If a method's return type is Void we Can have return Statement inside that but that method should not return any Value.

ex:
```
public static Void test()
{
  s.o.pln("Hello");
  return;
}
```

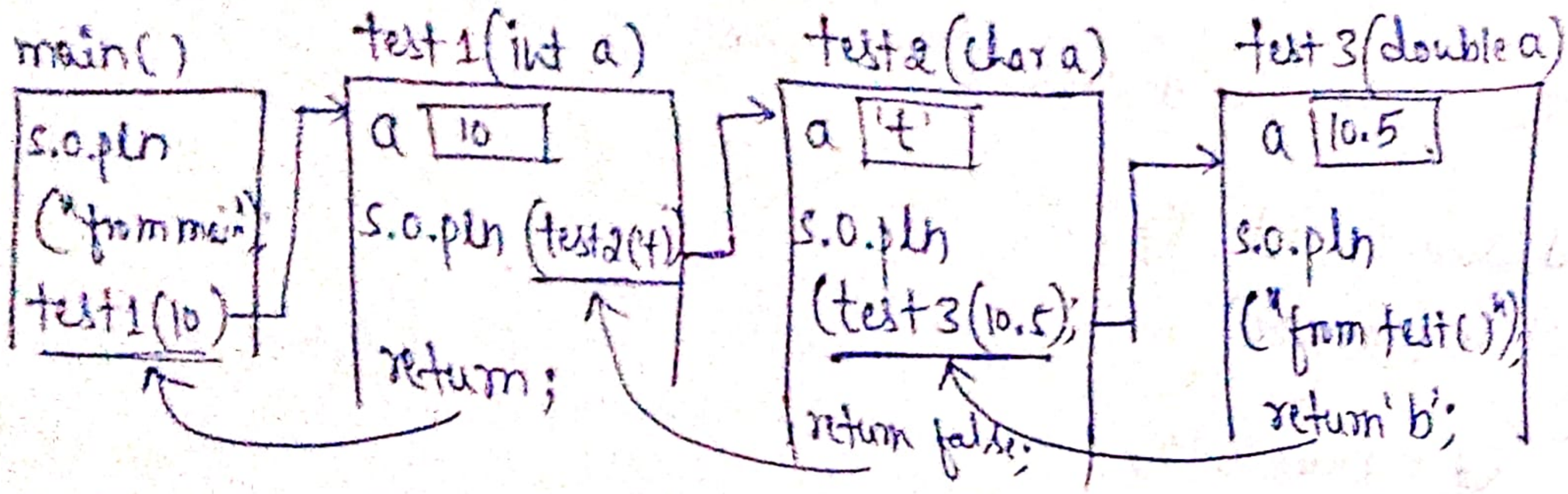
O/P

Hello.

```
class A
{
    public static void test1(int a)
    {
        s.o.pln(test2('t'));
        return;
    }

    public static boolean test2(char a)
    {
        s.o.pln(test3(10.5));
        return false;
    }

    public static char test3(double a)
    {
        s.o.pln("from test3()");
        return 'b';
    }

    P.S.V.m(String[] args)
    {
        s.o.pln("from main()");
        test1(10);
    }
}
```

o/p

```
from main();
from test1()
from test3()
b
false
```



Note: * Return statement must be used as last statement of a block. If we write any statement after return in the same block will get compile time error.

eg:
```
class A
{
    public static void test1(int a)
    {
        return;
        s.o.pln("from test1()"); // CTE // Unreachable statement error.
    }
    P.S.V.m(String[] args)
    {
        test1(10);
    }
}
```

* Wap to design a method to check whether a number is perfect number or not.
(a no. is said to be perfect no. if the sum of proper divisors of the number is equal to number itself)

```
class Perfect
{
  p.s.v.m (String[] args)
  {
    int result = test(6);
    System.out.println(result);
  }

  public static void ~~main~~ test (int num)
  {
    int sum=0;
    for (int div=1; div<=99; div++)
    {
      if (num%div==0 && sum!=num)
      {
        sum = sum+div;
      }
    }
    return sum;
  }
}
```

```
int num=6;
int sum=0;
for(int i=1; i<num; i++)
{
  if(num%i==0)
  {
    sum=sum+i;
  }
if(num==sum)
{
  S.o.println("perfect no");
}
else
{
  S.o.println("not Perfect")
}
}//end of for.
```

Method Overloading:

* A class having more than one method with same name but different formal arguments is called method "overloading".
* For method overloading name of the methods must be same but the formal arguments must be different. formal arguments can be diff in length, or order, or type.

ex

test()	test()	test(int a)	test(int a)	test(int a)
test()	test(int a)	test(double a)	test(int b)	test(int a, int b)
X	✓	✓	X	✓

test(int a, double b)
test(double a, int b)
✓

```
ex: class A
{
  public static void test()
  {
  }
  public static void test(int a)
  {
  }
  p.s.v.m (String[] args)
  {
    test(10);
```

ex' Class A

```
{
    public Static Void test1()
    {
        S.o.pln("from test1()");
    }
    public Static Void test1(int a)
    {
        S.o.pln("from test1(int)");
    }
    public Static Void test1(double a, int b)
    {
        S.o.pln("from test1(double, int)");
    }
    p.s.v.m(String[] args)
    {
        test1(10); // from test1(int)
        test1(); // from test1()
        test1(12.65, 25); // from test1(double, int)
        test1('a'); // from test1 ⇐ Widening
        test1(12.8); // CTE
    }
}
```

(*) WAP to perform a addition of 2, 3, 4 numbers with a proper msg. using return statement & Overloading method.

```
import java.util.Scanner;
class A
{
    p.s.v.m(string[] args)
    {
        Options();
    }

    public static Void Options()
    {
        Scanner s=new Scanner(System.in);
        s.o.pln("------ WELCOME ------");
        s.o.pln("Press 1 to add 2 numbers");
        s.o.pln("press 2 to add 3 numbers");
        s.o.pln("Press 3 to add 4 numbers");
        int num= s.nextInt();
        int resut;
```

```
if(num==1)
{
    s.o.pln("Enter 2 numbers");
    int a= s.nextInt();
    int b= s.nextInt();
    result = add(a,b);
    s.o.pln("Addition of 2 no's:" +result);
    msg();
}
else if(num==2)
{
    s.o.pln("Enter 3 numbers");
    int a= s.nextInt();
    int b= s.nextInt();
    int c= s.nextInt();
    result = add(a,b,c);
    s.o.pln("Addition of 3 no's:" +result);
    msg();
}
else if(num==3)
{
    s.o.pln("Enter 4 numbers");
    int a= s.nextInt();
    int b= s.nextInt();
    int c= s.nextInt();
    int d= s.nextInt();
    result = add(a,b,c,d);
    s.o.pln("Addition of 4 no's:" +result);
    msg();
}
else
{
    ~~msg();~~ endmsg();
}
}// end of options
public ~~void~~ static void endmsg();
{
    s.o.pln("----THANK YOU----");
}
```

```
public Static Void msg()
{
    Scanner s = new Scanner (System.in);
    S.O.pln ("Press 1 to Continue");
    S.O.pln ("Press any other number to Stop");
    int a = S.nextInt();
    if (a == 1)
    {
        options();
    }
    else
    {
        endmsg();
    }
}

public Static int add (int a, int b)
{
    int Sum = a+b;
    return Sum;
}

public Static int add (int a, int b, int c)
{
    int Sum = a+b+c;
    return Sum;
}

public Static int add (int a, int b, int c, int d)
{
    int Sum = a+b+c+d;
    return Sum;
}

}// end of class.
```

Recursive Methods:-

* A method calling itself is called Recursive Method.
* When a method calls itself all the statements of that method will be repeatedly executed for infinite times.

ex: Class A

```
{
  public static void test()
  {
      s.o.pln("from test()");
    } test();  <= Calling itself
public static void main(String args)
{
  s.o.pln("from main");
  } test();
}
```

O/P

```
from main()
from test()
from test()
from test()
:
```

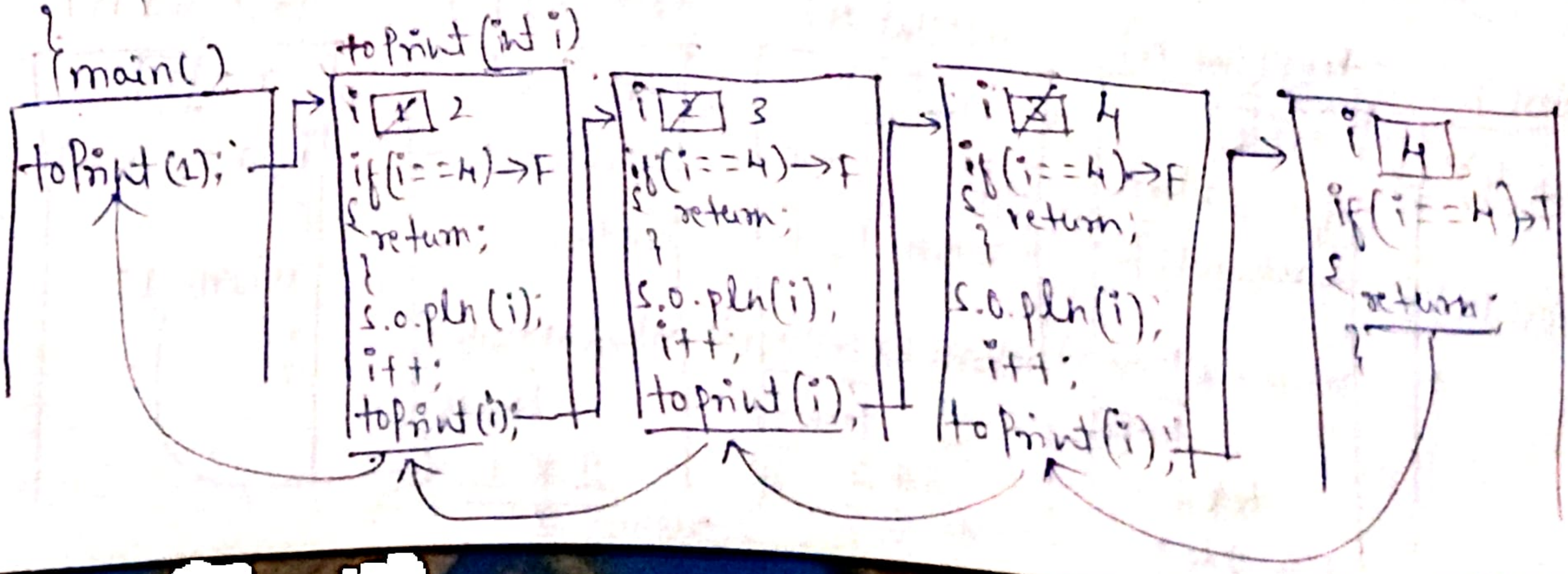
* In above example test() calls itself repeatedly & it won't stop calling therefore it becomes an infinite method calling.
* We can stop Recursion using "Base Case Conditions"

```
Class A
{
  public static void toPrint(int i)
  {
    if(i==4)
    {
      return;
    }
    s.o.pln(i);
    i++;
    toPrint(i);
  }
  p.s.v m(String[] args)
  {
    toPrint(1);
  }
}
```

Base Case Condition i.e to stop the recursion

O/P

```
1
2
3
```

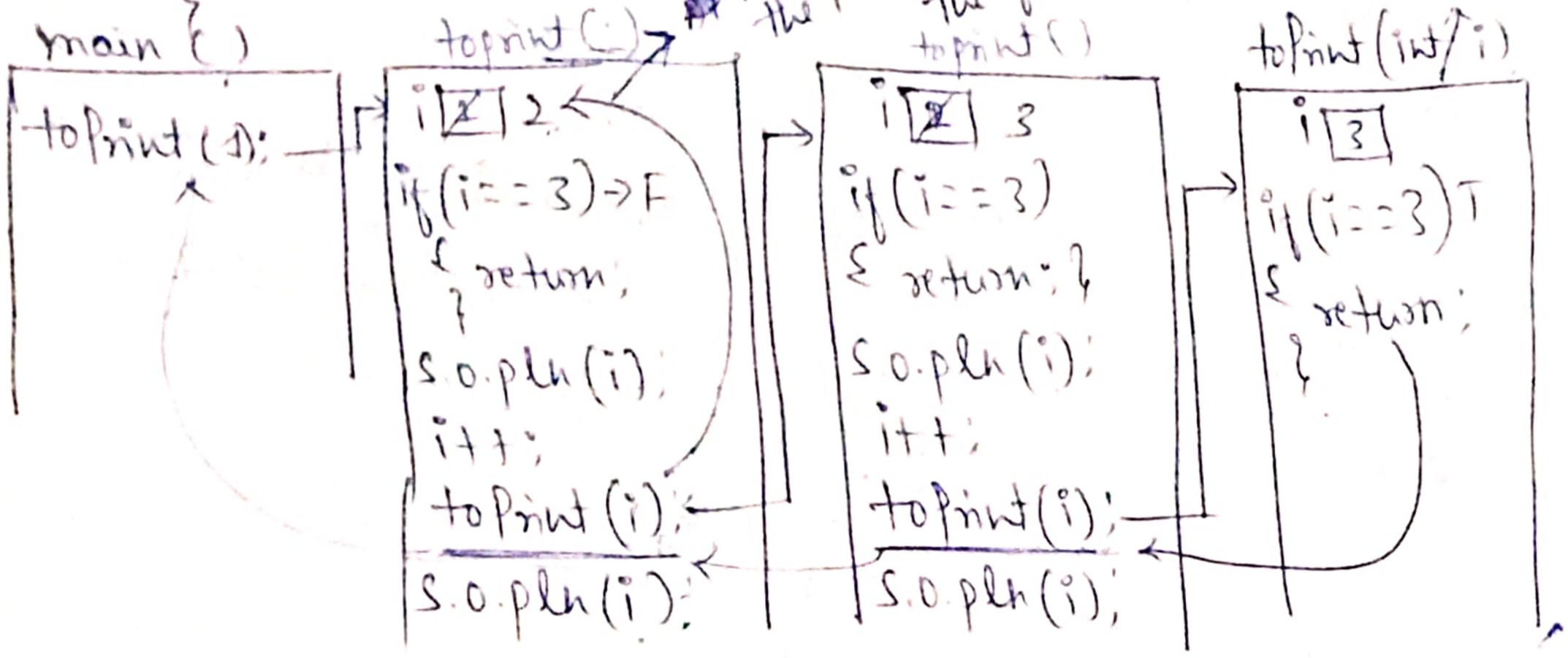


```
class
{
  public Static Void toPrint(int i)
  {
    if(i==3)
    {
      return;
    }
    S.o.pln(i);
    i++;
    toPrint(i);
    S.o.pln(i);
  }

  p.s.v m(String[] args)
  {
    toPrint(1);
  }
```

i [1] 2 3

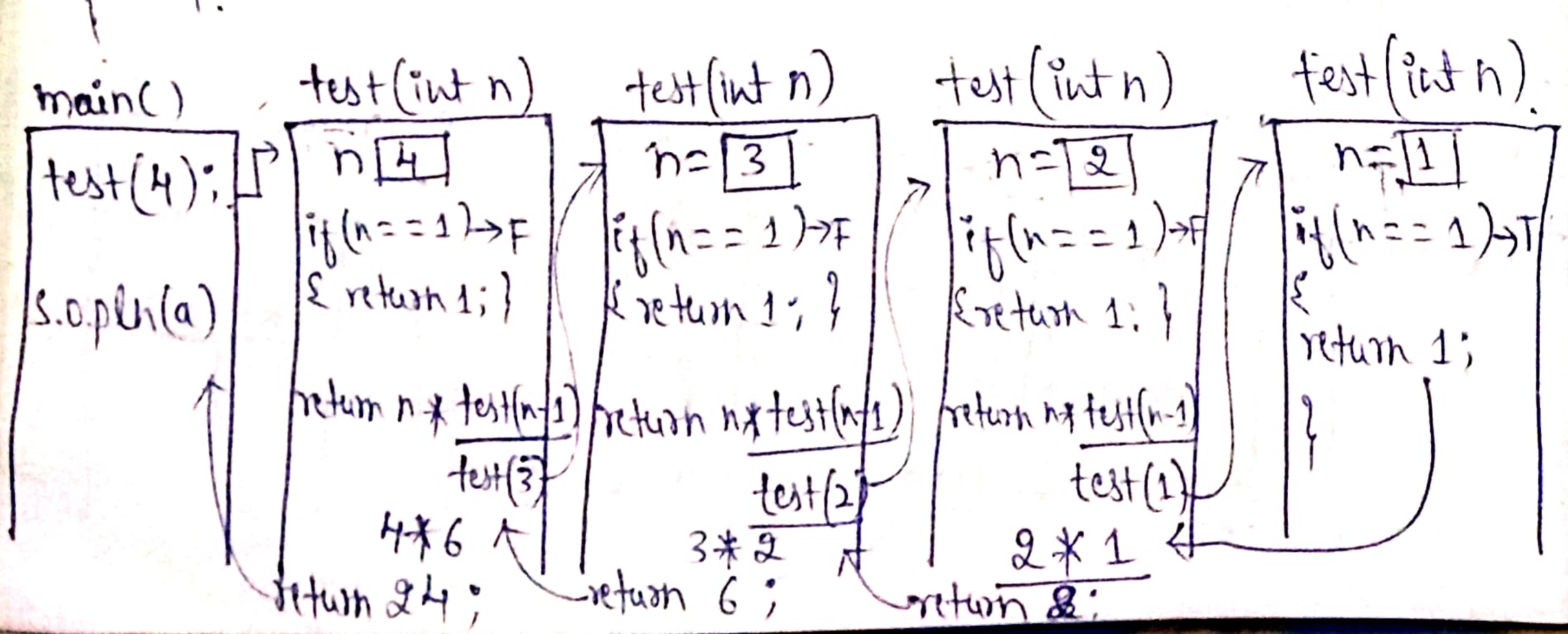
o/p: 1 2 3 2

o/p: 1 2 3 2




```
ex: class S
  {
    public static int test(int n)
    {
      if(n==1)
      {
        return 1;
      }
      return n*test(n-1);
    }
    public Static Void main(String[] args)
    {
      int a=test(4);
      s.o.pln(a);
    }
}
```

o/p: 24



*NAP to print only even no's from 1 to 50 without using looping statement.

```
class A
{
public static void main(string[] args)
{
  even(1,50);
}
public static void even(int a, int b)
{
  if(a==b)
  {
    s.o.pln(a);
    return;
  }
  else if(a%2==0)
  {
    s.o.pln(a);
  }
  a++;
  even(a,b);
}//end of even
}//end of class.
```

```
(a==b)
if(a>b)
{
  return;
}
else if(a%2==0)
{
  s.o.pln(a);
}
a++
even(a,b);
}// end of even method.
}// end of class
```

Array :-

* Array is a Continues block of memory which is used to Store multiple Values of Same type (Homogenious Values).

* Once we Create array we will get address or reference of the array in return & we have to Store their reference inside array reference Variable.

How to Create Array Reference Variable :

* The Variable Created in Order to Store address or reference of the array is Called array reference Variable.

Syntax: datatype [ ] Variable;

```
int [ ] a;        (or)  int a[];
double [ ] b;           double b[];
boolean [ ] c;          boolean c[];
```

Note: Array reference Variable Can Store

(i) Address or reference of the array.

(ii) Null

* If the array reference Variable is initialized with null it means it is not refering to any array.

How to Create Array :

* We Can Create array in 2 Ways.

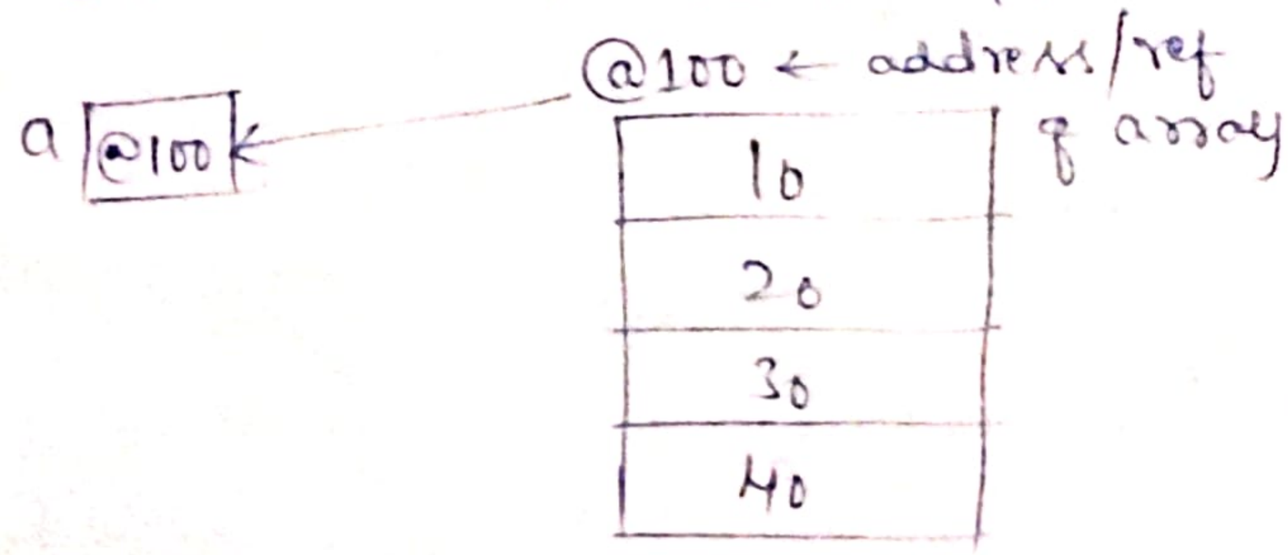
(i) Using declaration & initialization Statements

(ii) Using "new" Keyword.

(i) Using declaration & initialization Statement

Syntax: datatype[ ] Variable = { Value1, Value2, Value3 - - }

ex: int [ ] a = { 10, 20, 30, 40 };



ex: class A
{
p.s.v m(String[] args)
{
double [] a = {12.5, 6.8, 5.9};
s.o.pln(a); // reference/address of array
}
}

How to find length of an Array:

* Length of the array is always equal to no. of values stored inside it.
* We can fetch the length of an array with the help of Predefined variable called "Length".

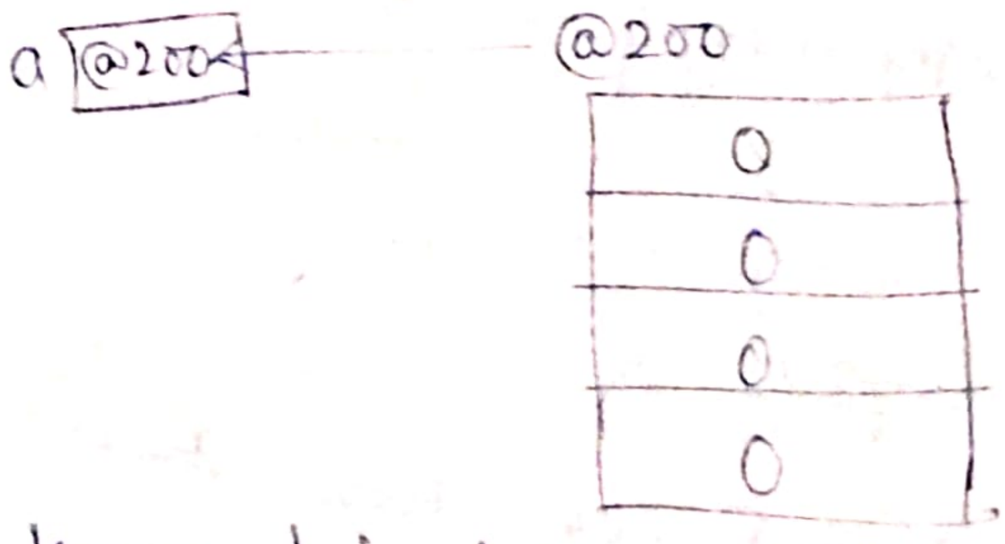
Syntax: arrayref Variable. Length

ex: class A
{
p.s.v m(String[] args)
{
int [] a = {1, 2, 3, 4};
s.o.pln("Lenth of an array is: " + a.length);
}
}

(ii) Creating Array Using "new" Keyword:

Syntax: datatype [] Variable = new datatype[size];

ex: int [] a = new int [4];



* "new" is a Keyword in java which is used to create an array or objects.
* When we create array using new keyword initially it will have the default values of the given data type.

Assigning (or) Storing Values to Array.

* We can assign or access values from the array with help of "index / Subscript".
* Index always starts from '0' & end's at "Length of array - 1".

int [] a = new int [4];

a [@100] ← @100

0	0
1	0
2	0
3	0

Length → 4

Syntax to assign values to array

Array ref Var [index] = Value;

ex:
a[0] = 10;
a[1] = 20;
a[3] = 30;

a [@100] ← @100

0	10
1	20
2	0
3	30

Length → 4

Syntax to access the values of array

Array ref Variable [index];

ex:
S.o.pln (a[0]); //10
S.o.pln (a[1]); //20
S.o.pln (a[3]); // 30.

ex: WAP to Create character array of size 5, Store any 5 chars & Print them one by one.

```
class A
{
    p.s.v.m(String[] args).
    {
        ~~char a[0] ch[7]~~ = char [] ch = new char[5];
        ch[0] = 'a';
        ch[1] = 'b';
        ch[2] = 'c';
        ch[3] = 'd';
        ch[4] = 'e';

        S.o.pln (a[0]);
        S.o.pln (a[1]);
        S.o.pln (a[2]);
        S.o.pln (a[3]);
        S.o.pln (a[4]);
    }
}
```

(or)

```
for(int i=0; i<a.length ; i++)
{
    S.o.pln(a[i]);
}.
```

* WAP to find sum of all the no's present in a integer array.

```
class A
{
    p.s.v m (String[] args)
    {
        int sum=0;
        int [] a = { 10,20,30,40,50};
        for(int i=0; i<a.length; i++)
        {
            sum = sum+a[i];
        }
        s.o.pln ("the sum of all the no's " + sum);
    }
}
```

* WAP to find largest number in a integer ~~value~~ array.

```
class A
{
    p.s.v m( String [] args)
    {
        int [] a = { 10, 20, 30, 40, 50};
        int largest = a[0];
        for(int i=1; i<a.length; i++)
        {
            if(a[i]>largest)
            {
                largest = a[i];
            }
        }
        s.o.pln (largest);
    }
}
```

Array Index Out of Bound Exception:

* If the index used is equal or greater than length of the array. We will get array index out of bound exception..

ex:

```
class A
{
    p.s.v.m(String[] args)
    {
        int [] a = { 1,3,5,7};
        s.o.pln (a[0]); // 1
        s.o.pln (a[3]); // 7
        s.o.pln (a[a.length]); // Array Index Out of Bound Exception
    }
}
```

o/p

1

7

Array Index Out of Bound Exception ← Run time error

* Null Pointer Exception

* when ever the array ref variable as initialized with a null if we try to use it for accessing or assigning the values to the array we will get null pointer exception.

ex. Class A

```
{
  p.s.v.m(S[] a)
  {
    double [] a = null;
    a[0] = 10.5 ; // Null pointer Exception
  }
}
```

* Write a pgm to Create an integer array by taking size & values from the user & also print the values one by one.

```
import java.util.Scanner;
Class A
{
public static Void main(String[] args)
{
  Scanner s = new Scanner(System.in);
  s.o.pln("Enter the size of the array");
  int siz = s.nextInt();
  Int [] a = new Int [siz];
  for(int i=0; i<=siz; i++)
  {
```

~~* wap to generate 1st 10 odd nos from then with an array & print the values one by one.~~

```
    if(i<siz)
    {
      s.o.pln("Enter the Value for a[" + i + "]");
      Int Vl = s.nextInt();
      a[i] = Vl;
    }
  }
  for(Int i=0; i<a.length; i++)
  {
    s.o.pln("Array Value of a[" + i + "]" is " + a[i]);
  }
}
```

* WAP to generate 1st 10 odd numbers. Store them inside an array & print the values one by one.

```
class
{
  p s v m (s[] args)
  {
    int [] a = new int [9];
    int n = 1;
    for(int i=0; i<a.length; i++)
    {
      if (n%2==1)
      {
        a[i]=n;
        s.o.pln("first ten odd nos are:"+a[i]);
      }
      i++;
    }
  }
}
```

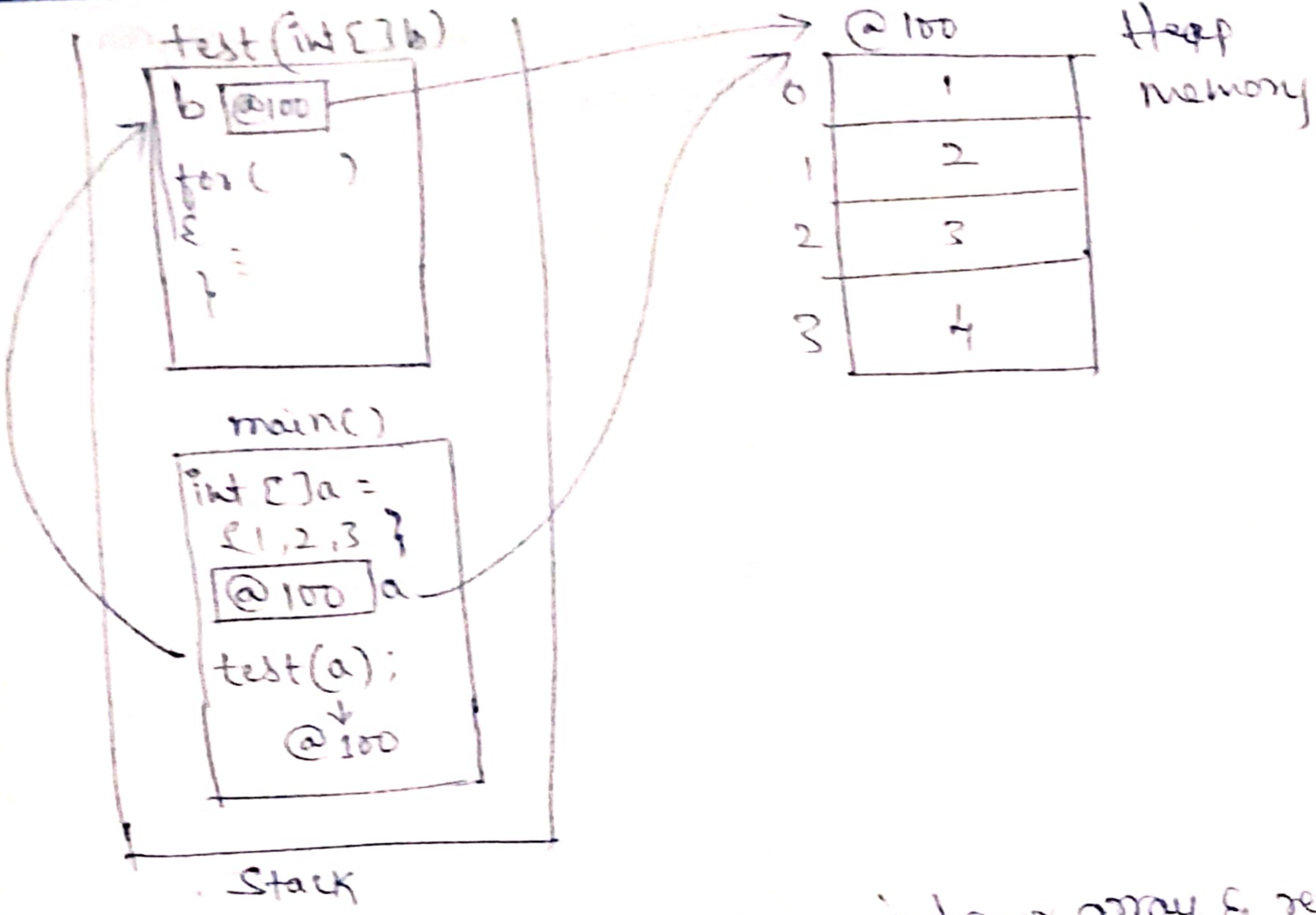
Passing an Array to a method

* The array always gets created in the heap memory.

ex: Class A

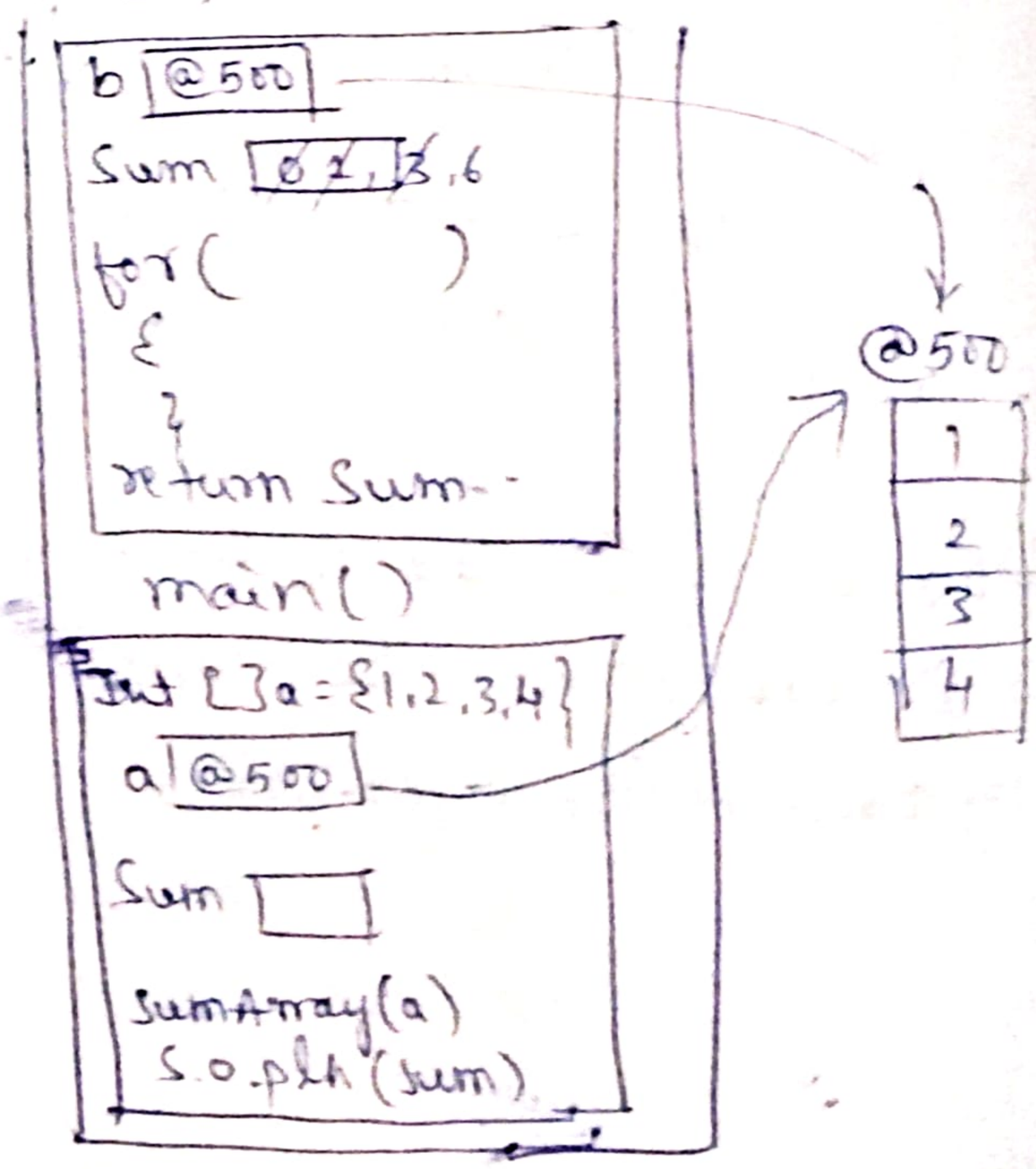
```
{
  public static void test (int []b)
  {
    for(int i=0; i<b.length; i++)
    {
      s.o.pln(b[i]);
    }
  }
  p s v.m (String[] args)
  {
    int []a = {1,2,3,4}.
    test(a);
  }
}
```

* The array created in the heap memory will have global scope & we can access the values of the array from any method by passing the reference.



* Design a method which accepts a integer array & returns sum of all the passing no's.

```
class A
{
  p.s.v.m(String[] args)
  {
    int []a = {1,2,3,4};

    int Sum = SumArray(a);

    S.o.pln(Sum);
  }

public static int SumArray(int []b)
{
  int Sum=0;
  for (int i=0; i< b.length; i++)
  {
    Sum = Sum+ a[i];
  }
  return Sum;
}
}
```



② Design a method which accepts a integer array & return Count of even no's from the array.

```
class A
{
  p.s.v.m(s[]args)
  {
    int a[]= {1,2,3,4,5,6};
    int Count = Enumber(a);
    S.o.pln("Even No's in array are" + Count);
  }

  public static int Enumber(int []b)
  {
    int Count=0;
    for(int i=0; i< b.length; i++)
    {
      if (~~b%2==0~~ (b[i]%2==0)
      {
        Count++;
      }
    }
    return Count;
  }
}
```

* Design a method which accepts a integer array & return from smaller array from the array.

```
class A
{
  p.s.v.m(s[] args)
  {
    int a[]= {20,40,30,10,50};
    int Smallest = Smaller(a);
    S.o.pln("The Smallest No. in array is :"+Smallest);
  }
  public static int Smaller(int [b b)
  {
    int SmallNo = b[0];
    for(int i=0; i< b.length; i++)
    {
      if(b[i]< SmallNo)
      {
        SmallNo = b[i];
      }
    }
    return SmallNo;
  }
}
```

* WAP to check whether 'a' is present in given array.

```
class A
{
    p.s. ~~void~~ char charfind(char [] b)
    {
        char ch = b[0];
        for (int i=0; i<b.length; i++)
        {
            if (b[i] == ch)
            {
                ch = b[i];
            }
        }
        return ch;
    }
    public s.v.m (String [] args)
    {
        char[] a = {'a','e','i','o','u'};
        char result = char(a);
        if (result == 'a')
        {
            s.o.pln("char a is present");
        }
        else
            s.o.pln("char a is not present");
    }
}
```

```
*{ p.s.v.m( )
  {
    char [] ch = {'p','q','r','s','a'};
    char tofind = 'a';
    boolean flag = false;
    for (int i=0; i<ch.length; i++)
    {
        if (ch[i] == tofind)
        {
            s.o.pln("It is present");
            flag = true;
        } } break;
    if (flag == false)
    {
        s.o.pln("char is not present");
    } }
```

Command Line Arguments :

* Before Calling main method JVM internally Creates an array of String type & reference of this array Will be Stored inside formal argument that is 'args' of main method.
* The Size of the String type array Created by JVM Will be Zero if there is no Command line arguments passed. & if try to access anything from this array. It generates "array index out of bound exception".

ex:
```
class A
{
    public static Void main(String[] args)
    {
        S.o.pln (args); // Reference/address
        S.o.pln (args.length); // 0
        S.o.pln (args[0]); // ArrayIndexOut of bound Exception.
    }
}
```

Javac A.java ↵

Java A ↵ ⇐ No Command line Argument.

Note: If the Command line arguments are passed then length of the String array Created by JVM Will be Same as no. of Command line arguments passed.

ex:
```
class A
{
    public Static Void main(String[] args)
    {
        S.o.pln("Size of array" +args.length);
        for (int i=0; i<args.length ; i++)
        {
            S.o.pln (args[i]);
        }
    }
}
```

o/p

Javac A.java ↵

Java A Hi Hello Bye ↵ ← Command line argument

Size of array is 3

Hi

Hello

Bye.

eg: class A

```
{
  p.s.v.m(string[] args)
  {
    s.o.pln(args[0]+args[1]); //1020
  }
}
```

Java A 10 20 ↵

1020

* WAP to take 5 names as Command line arguments & Print them in reverse order.

Global Variables:

* The variable that are declared directly inside the class are called global variables.
* Global variables can be used through out the class
* Global variables take default values therefore initialization is not mandatory.

* 2 type of Global variables

(1) Static Variable
(2) Non-Static Variable.

Static members:

* Members of java pgm such as variable, block, & method which are prefix with static keyword are called "static members"

(i) Static Variable
(ii) Static block
(iii) Static Method

} Static Members

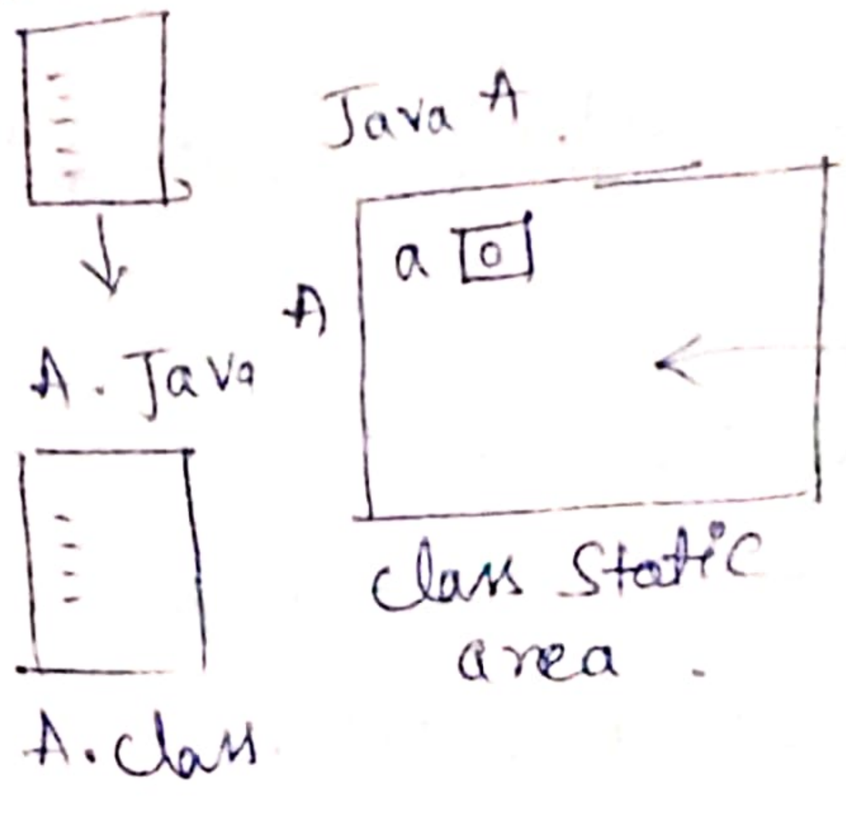
(i) Static Variable:

* A variable declared directly inside the class with static keywords prefix to it is called static variable
* The static variable gets their memory insert class static area
* Static variables can be use throughout the class since static variables are global variables they ~~take~~ take default values therefore initialize are not mandatory

Note: whenever the static variable name & local variable name are same highest priority to local variable in this case if we want to access static variable values we should make use of class name.

Syntax: `Class name.Variable`

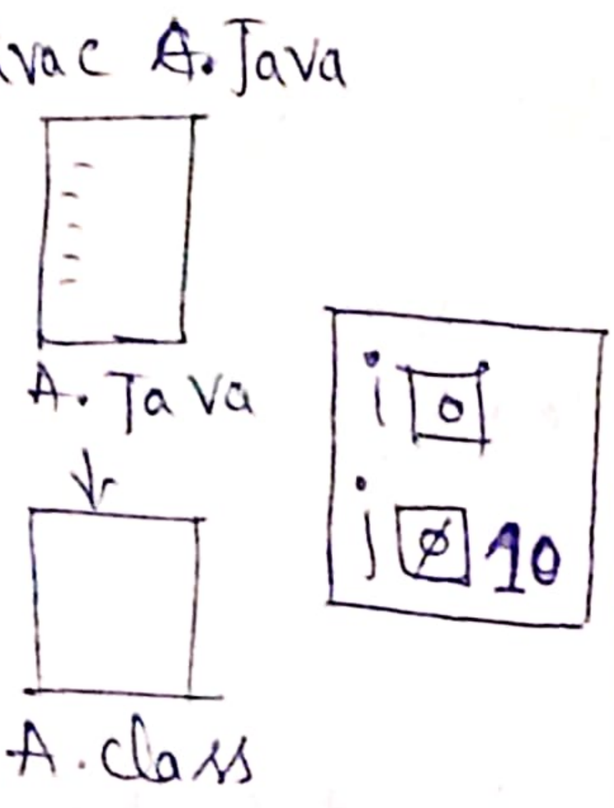
```
class A
{
  static int a;
  p.s.v.m(S[] args)
  {
    s.o.pln(a); //0
  }
}
```



* static Variable Can be used in two ways inside main method.

(i) Directly
(ii) using class name.

ex:
```
class A
{
  static int i, int j;
  p.s.v.m(S[] args)
  {
    s.o.pln(i); //0
    s.o.pln(A.j); //0
    j=20;
    s.o.pln(j); //20
    s.o.pln(A.j); //20
    test();
    int j=40;
    s.o.pln(j);
    s.o.pln(c.j);
    s.o.pln(i);
    s.o.pln(A.i);
  }
  public Static Void test()
  {
    int i=10;
    s.o.pln(i);
    s.o.pln(c.i);
    s.o.pln(j);
    s.o.pln(c.j);
    i=30;
  }
```

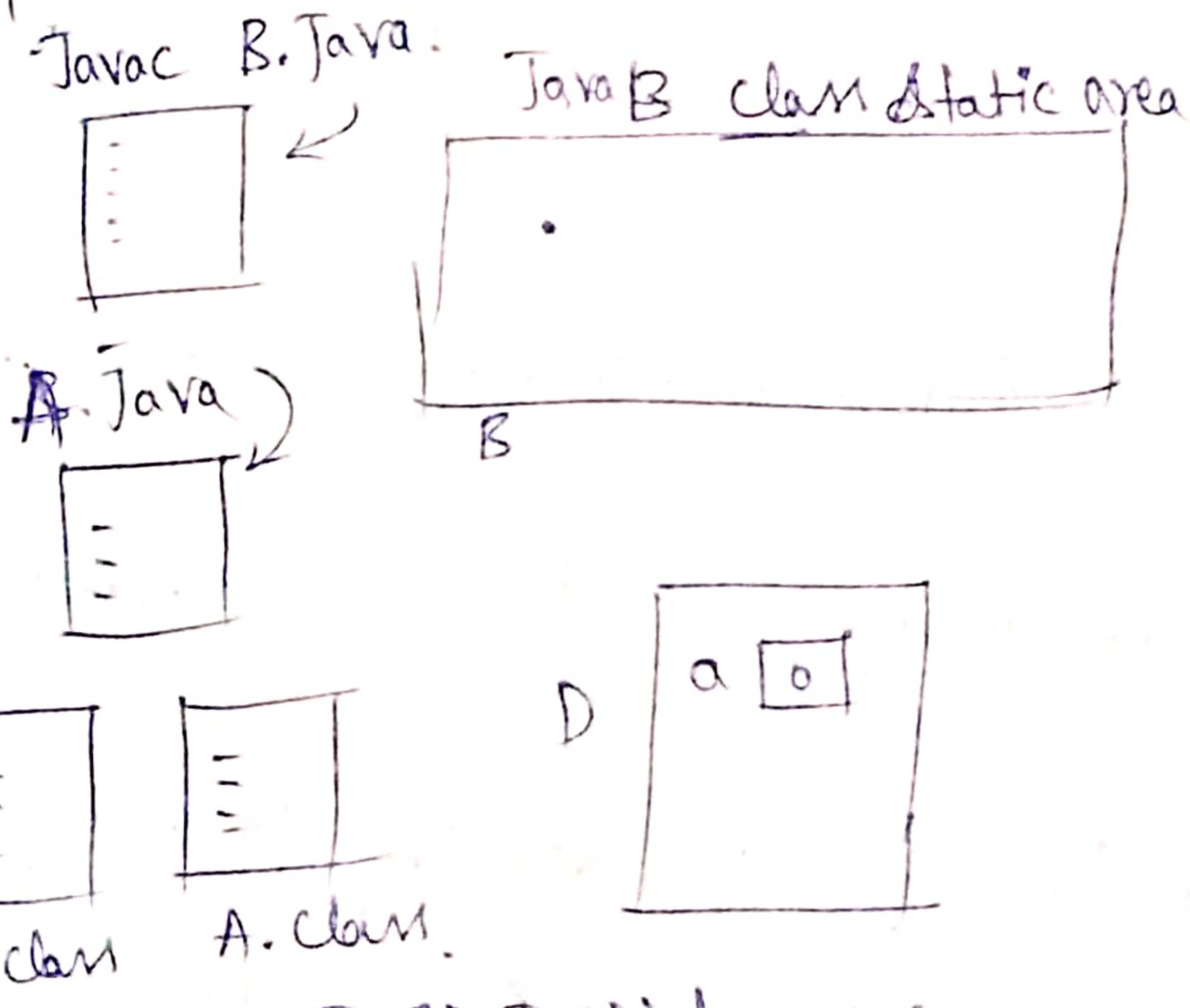


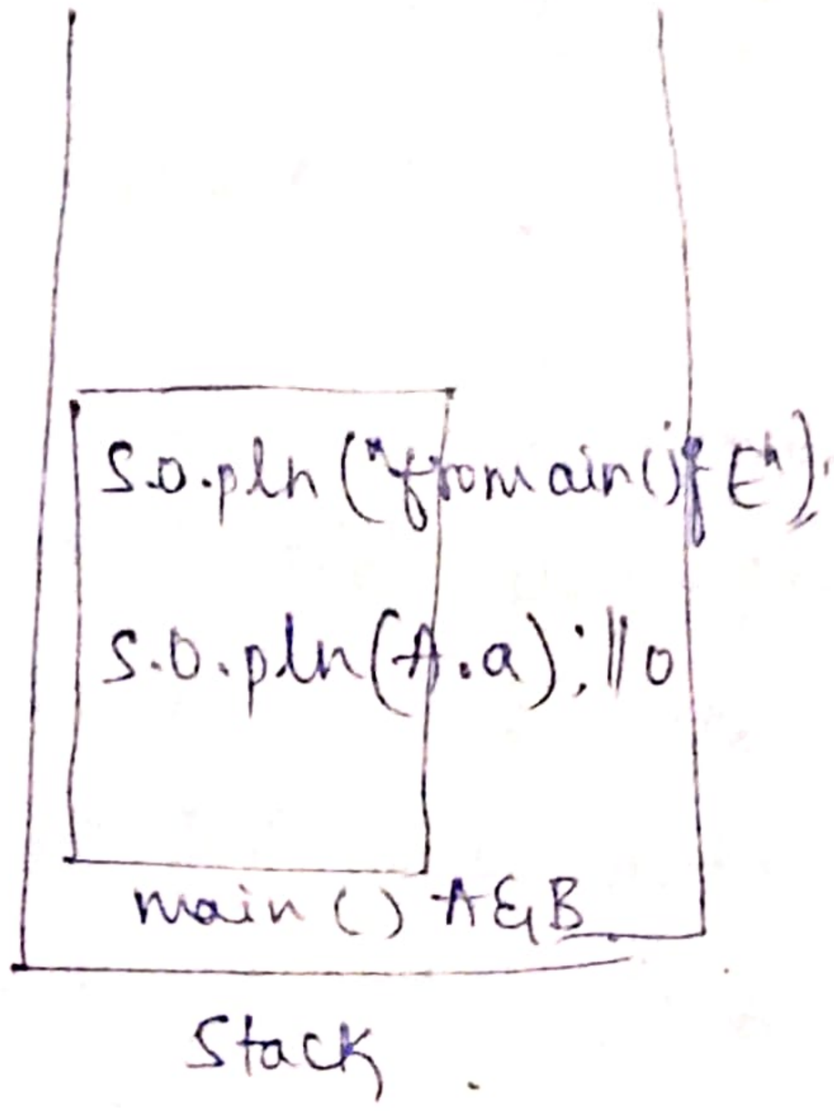
o/p
0
0
20
20
10
0
20
20
40
20
0

Note: We can use static Variable of one class in another class with the help of class name as shown in below example.

```
class A
{
  int a;
}

class B
{
  p.s.v.m(string[] args)
  {
    s.o.pln("from main() of E");
    s.o.pln(A.a); //0
  }
}
```

Javac B.Java.



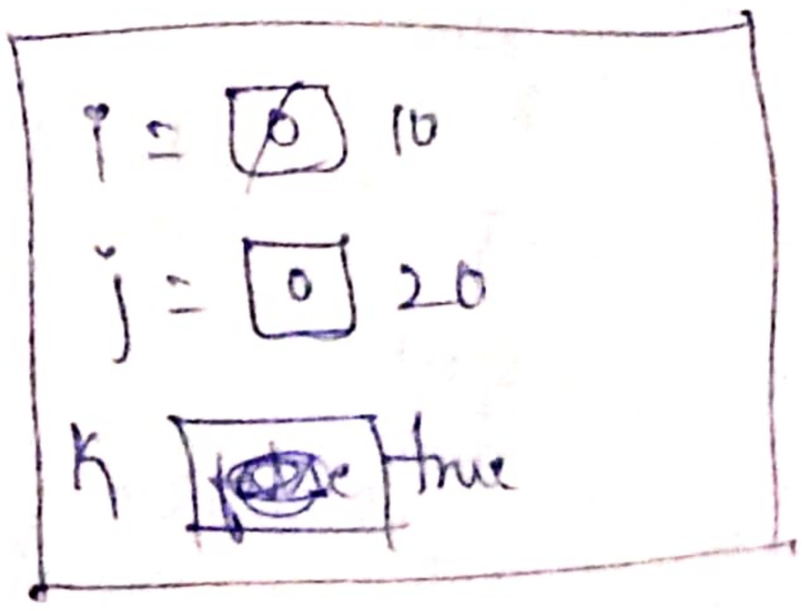


Static Initializer :-

ex:
```
class A
{
  static int i=10;
  static int j=20;
  static boolean k=true;
  p.s.v.m(S[] args)
  {
    s.o.pln(i); //10
    s.o.pln(j); //20
    s.o.pln(k); //true
  }
}
```

o/p
10
20
true

Class Static Area



Note: The static Variables will be always initialize with default Values first then the static initializers will be executed in top to bottom order.

* In the above example i=10, j=20; k=true are static initializer & get executed in top to bottom order.

(ii) Static Block :-

* A block prefixed with static keyword is called static block
* Static block are type of static initializers.
* Static blocks are also called as static initializer block

Syntax:
```
static
{
    statements;
}
```

* Static blocks does not have name, therefore we can't explicitly call them.
* Static blocks don't have return type therefore it can't return any value.
* Static blocks executed during class loading process.
* We can have multiple static blocks in a class they gets executed in top to bottom order.

ex:
```
class A
{
    static
    {
        s.o.pln("From Static-1");
    }
    static
    {
        s.o.pln("From Static-2");
    }
    p.s.v m(String[] args)
    {
        s.o.pln("main() starts");
    }
}
```

o/p
```
From Static-1
From Static-2
main() starts.
```

Note:- A Variable declared inside the static block will be a local Variable of that block

ex:
```
class A
{
    static
    {
        int a; //Local Variable
    }
    p.s.v.m(S[] args)
    {
        S.o.pln(a); //CTE
    }
}
```

ex
```
class A
{
    static int a;
    static int b;

    static
    {
        S.o.pln("From Static-1");
        S.o.pln(a);
        S.o.pln(b);
        a=10; //No local Variable Called 'a'
              so a will be initialized
    }
    Static
    {
        S.o.pln("From Static-2");
        S.o.pln(a);
        S.o.pln(b);
        b=20;
    }
    p.s.v.m(S[] args)
    {
        S.o.pln("From main()");
        S.o.pln(a);
        S.o.pln(b);
    }
}
```

o/p

```
From Static-1
0
0
From Static 2
10
0
From main()
10
20.
```

ex class L

```
{
    static int i=10;
    static int j=20;
    static
    {
      s.o.pln("from SIB-1 of L");
      s.o.pln(i); //10
      s.o.pln(j); //20
      j=100;
    }
    p.s.v.m(String[] args)
    {
      s.o.pln("from main() of L");
      s.o.pln(i); //10
      s.o.pln(j); //100.
    }
}
```

Class M

```
{
    static int K;
    static
    {
      s.o.pln("from SIB-1 of M");
      K=200;
    }
    p.s.v.m(s[] args)
    {
      s.o.pln("from main() of M");
      s.o.pln(K); //200.
      s.o.pln(L.i); //10 ⇐ only static block will be executed including static variables. "main" method will not be executed (i.e L's main).
      s.o.pln(L.j); //100.
    }
}
```

o/p

From SIB-1 of M
From main() of M
200
From SIB-1 of L
10
20
10
100.

```
s.o.pln("from SIB-1 of L")
s.o.pln(i); → 10
s.o.pln(j); → 20
j=100;
```

static of L

```
s.o.pln("from main() of M");
s.o.pln(K);
s.o.pln(L.i); → 10
s.o.pln(L.j); → 100
```

Main() of M

```
s.o.pln("from SIB-1 of M")
K=200
```

static block of M.

Javac M.java

M.Java

Java M

class static area

M.class L.class

L.java

M | K ~~0~~ 200

L | i ~~0~~ 10
j ~~0~~ ~~20~~ 100

Stack

* In above example we are using members of class 'L' inside class M therefore class L becomes dependent class for class M.

* When we compile M.java the dependent class L also gets compiled therefore we will get two class files i.e M.class & L.class

* When we execute class M during the execution of following instn

System.Out.Println(L.i); class L gets loaded.

During the execution of the next instruction i.e

System.Out.Println(L.j); since class L is already loaded it will not be loaded again.

* Can we execute an instn before execution of main()?

* Yes we can execute an instn before execution of main method with the help of "Static Initializers".

ex. class A

```
{
  static int a = test();
  p.s. int test()
  {
     s.o.pln("from test()");
     s.o.pln(a);
     return 10;
  }
  public static void main(S[] args)
  {
     s.o.pln("from main()");
     s.o.pln(a);
  }
}
```

O/P

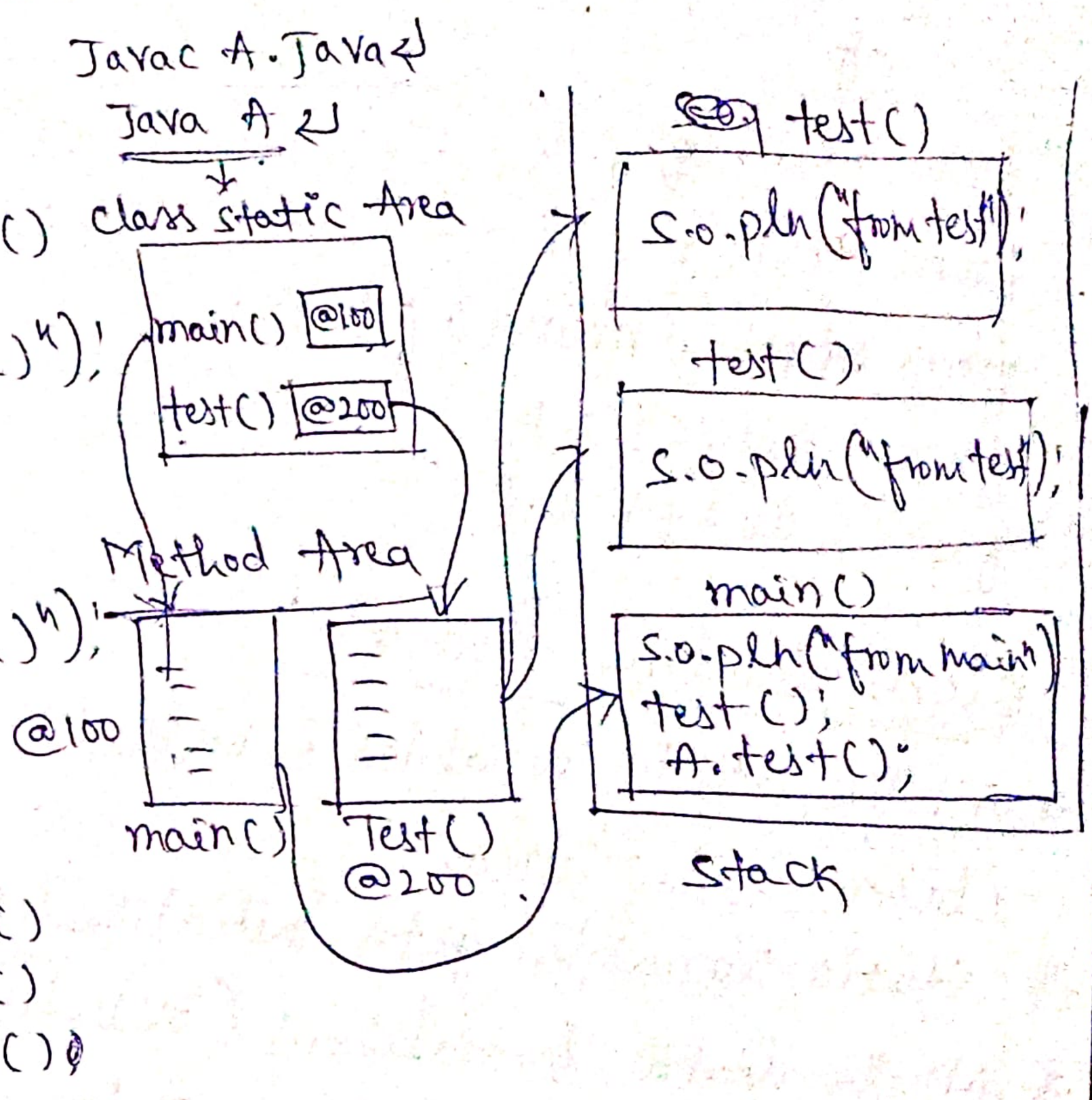
```
From test()
0
from main()
10
```

Static Method

* A method prefix with static keyword is called "static method"
* static method body will be stored in "method area" & it's reference will be stored in "class static area".
* static methods can be used in two ways
  (i) Directly
  (ii) With the help of class name.

ex:
```
class A
{
public static void test()
{
  s.o.pln("from test()");
}
p.s.v.m(S[] args)
{
s.o.pln("from main()");
test();
A.test();
}
}
```

o/p
from main()
from test()
from test()



* We can call static method of one class in another class with help of class name.

ex:
```
class P
{
  public static void test()
  {
    s.o.pln("from test() of P");
  }

  p.s.v.m(String[] args)
  {
    s.o.pln("from main() of P");
    test();
    P.test();
  }
}
```

```
class Q
{
  public static void main(String[] args)
  {
    S.o.pln("from main() of Q");
    P.test(); // Calling test method in P class
    P.main(null); // Calling main method in P class with
                  "null" argument (we can give String also)
  }
}
```

o/p

```
from main() of Q
from test() of P
from main() of P
from test() of P
from test() of P
```

Class Loading Process:

* All the static variables & methods are loaded in the class static area.
* Static variables gets initialized with default value
* All the static initializers (including static blocks) gets executed in top to bottom order.

Conclusion:

* All the static variables & methods of the class can be used directly or with the help of class name inside the same class.
* All the static variables & methods of one class can be used in another class with the help of class name

diff Static block Static Method

```
class StaticDemo
{
static int a;
static int b;
static
{
    s.o.pln("SIB-1 starts...");
    s.o.pln(a);
    s.o.pln(StaticDemo.b);
    StaticDemo.a=10;
    b=20;
    s.o.pln("SIB-1 ends...");
}
static
{
    s.o.pln("SIB-2 starts...");
    s.o.pln(a);
    s.o.pln(b);
    b=30;
    s.o.pln("SIB-2 ends...");
}
public static int test()
{
    s.o.pln("test() starts...");
    s.o.pln(a);
    s.o.pln(b);
    s.o.pln("test() ends...");
} return 50;
}.
```

```
p.s.v m(S[] args)
{
    s.o.pln("main() starts...");
    int a=test();
    s.o.pln(a);
    s.o.pln(b);
    s.o.pln("main() ends...");
}
}.
```

Non-Static Members :-

(i) Non-Static Variables
(ii) Non-Static block
(iii) Non-Static Method
(iv) Constructors.

* All the non-Static Variables & non-Static methods of the class Will be loaded in the object of the class.
* therefore before using non-Static Variable (or) non-Static method we must Create object for the class
* Object of the class is also Called as "instance", the process of Creating object of the class is Called instanciation.

How to Create Object for the Class:

* Object is a block of memory Created in the heap area & it stores non-static Variables & non-Static methods of the class.
* We Can Create object with the help of "new" Keyword & "Constructor" of the class.
* Syntax to Create an object

```
new Constructor;
```

* When we execute above statement an object Will be Created in the heap memory & all the non-Static Variables & methods of the class Will be loaded inside it.
* The object Created will have some reference/address & we have to Store this reference in Variable.
* The Variable Created in order to Store address/ref of the obj is Called "object reference Variable".
* Syntax to Create object reference Variable

```
class name Variable name;
```

* In the above syntax classname acts as non-primitive data type.
* The object reference variable can store.
  (i) Reference of the object created.
  (ii) Null.

* Complete syntax to create object & store the reference.

```
Classname Variablename = new Constructor;
```

Constructor :-

Constructor is a special non-static member of a class which helps us to create an object.

Syntax: Access Specifier class_name ([datatype Var1, datatype Var2 ...])
{
Statements/Instructions;
}

* For constructors formal argument is not mandatory.

ex:
```
class A
{
  A()-
  {
    S.o.pln("From Constructor of A");
  }
  P.S.V.M(String[] args)
  {
    S.o.pln("from main()");
    A obj1 = new A();// Creating obj for class A
    S.o.pln(obj1);
  }
}
```

o/p
from main()
From Constructor of A

Note: If there is no Constructor is Created by Pgmer then Compiler by default adds a Constructor Without any arguments & it is Called as "Default Constructor".

Before Compile

```
class B
{
  p.s.v.m(S[] args)
  {
    s.o.pln("from main()");
    B obj1 = new B();
    s.o.pln(obj1);
  }
}
```

After Compile

```
Class B
{
  B() => Default Constructor
  {
  }
  p.s.v.m(S[] args)
  {
    s.o.pln("from main()");
    B obj1 = new B();
    s.o.pln(obj1); // reference.
  }
}
```

Types of Constructors

* There are 2 type of Constructors
  1. No-argument Constructor
  2. Parametirezed Constructor

1. No argument Constructor:

* A Constructor Without formal arguments is Called No-argument.

2. Parametirezed Constructor:-

* Constructors With formal arguments is Called Parametirized Constructor.

ex:

```
class C
{
  c(int a, int b) <= parametirized Constructor.
  {
    s.o.pln(a); // 10
    s.o.pln(b); // 20
  }
  p.s.v.m(S[] args)
  {
    s.o.pln("from main()");
    C obj1 = new C(10,20);
  }
}
```

Note: Compiler adds default Constructor inside the class only one there is no Constructor Created by the pgmer. If there is atleast one Constructor Created by the pgmer Compiler will not add default Constructor.

ex:
```
class D
{
    D(int a)
    {
        S.o.pln("From Constructor D(int)");
    }
    public S.V.m(String[] args)
    {
        S.o.pln("from main() of D");
        D obj = new D();// CTE
        S.o.pln(obj);
    }
}
```

Constructor Overloading:

* A class having more than one Constructor with different formal arguments is called Constructor Overloading.

ex:
```
class E
{
    E()
    {
        S.o.pln("From Constructor E()");
    }
    E(int a)
    {
        S.o.pln("From Constructor E(int)");
    }
    E(double a)
    {
        S.o.pln("From Constructor E(double)");
    }
    P.S.V.m(String[] args)
    {
        E obj1 = new E();
        E obj2 = new E(12.5);
        E obj3 = new E(10);
    }
}
```

o/p

From Constructor E()
From Constructor E(double)
From Constructor E(int)

(i) Non Static Variables :-

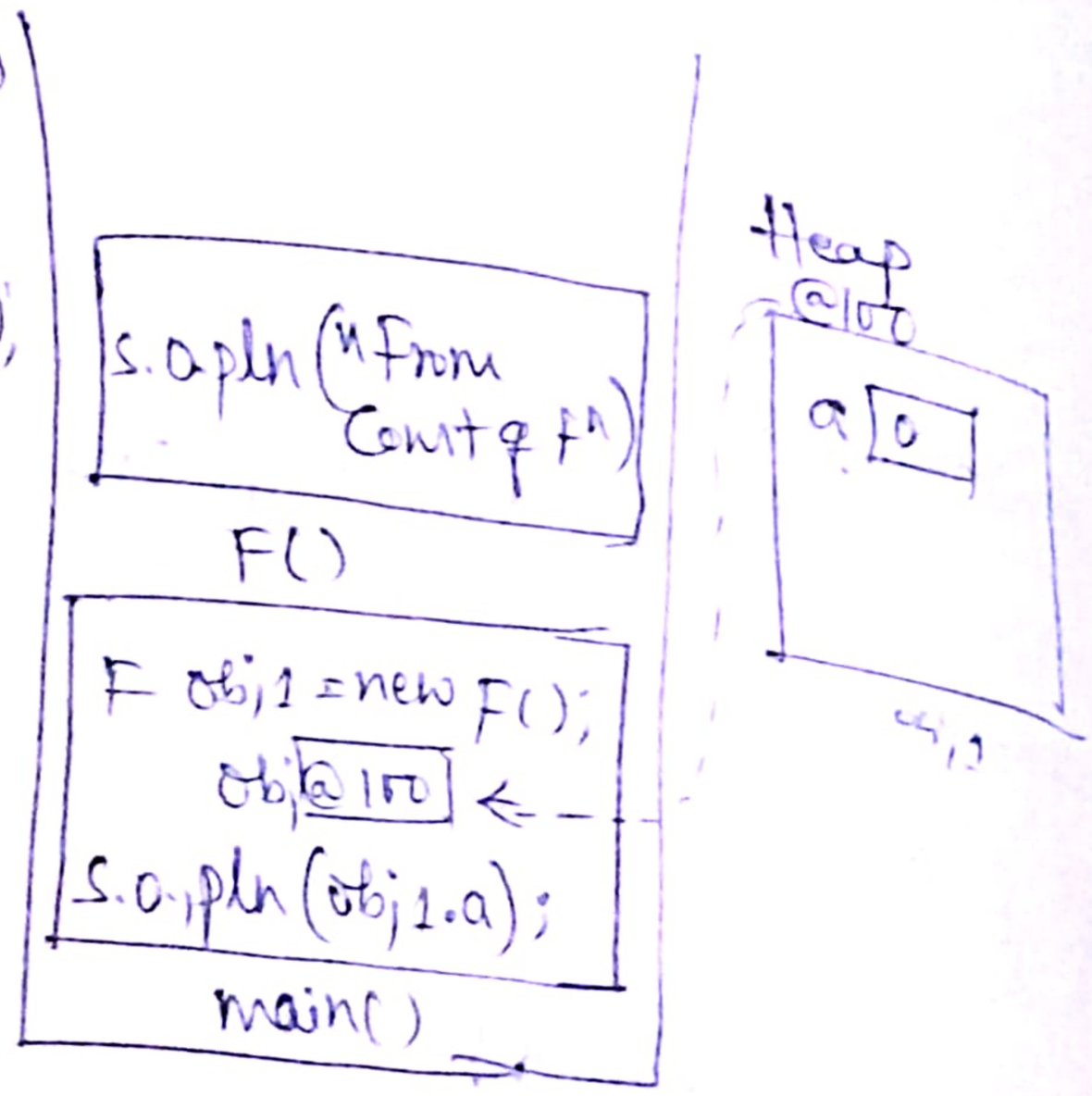
* Non Static Variables are global Variables which are declared directly inside the class without "Static" Keyword prefix
* Non Static Variables gets Stored inside Obj of the class
  ∴ before using we must Create object
* Since they are global Variables they Can be used throughout the class.
* Non-Static Variables takes default Values therefore initialization is not mandatory before using.

ex:

```
class F
{
  int a;
  F()
  {
    S.o.pln("From Constructor of F");
  }
  p.s.v.m(String[] args)
  {
    S.o.pln("from main()");
    F obj1=new F();
    S.o.pln(obj1.a);
  }
}
```

o/p

```
from main
from Const of F()
0
```



ex:

```
class G
{
  int a;
  int b;
  G()
  {
    S.o.pln("From G()");
  }
  p.s.v.m(String[] args)
  {
    G obj1 = new G();
    S.o.pln(obj1.a); //0
    S.o.pln(obj1.b); //0

    G obj2 = new G();
    S.o.pln(obj2.a); //0
    S.o.pln(obj2.b); //0

    obj1.a = 100;
    obj2.b = 200;

    S.o.pln(obj1.a); //100
    S.o.pln(obj1.b);
    S.o.pln(obj2.a);
    S.o.pln(obj2.b);
  }
}
```

o/p

```
From G()
0
0
From G()
0
100
0
0
200
```

ex: class H

```
{ int i;
  int j;
  p.s.v.m(S[] args)
  {
    H obj1 = new H();
    H obj2 = new H();

    obj1.i = 10;
    obj2.j = 20;

    obj2.i = 100;
    obj2.j = 200;
    s.o.pln(obj1.i);//10
    s.o.pln(obj1.j);//20
    s.o.pln(obj2.i);//100
    s.o.pln(obj2.j);//200
  }
}
```

op

10
20
100
200

Note: We can use non static variable of one class in another class by creating obj that class as shown in below example

ex:

```
class I
{
  int a;
}
```

ex:

```
class J
{
  p.s.v.m(S[] args)
  {
    I obj1 = new I();
    s.o.pln(obj1.a);
  }
}
```

Non-Static Initializers:

ex:
```
class K
{
    int a=10;
    int b=20;
    p.s v.m (String[] args)
    {
        K obj1 = new K();
        S.o.pln(obj1.a); //10
        S.o.pln(obj1.b); //20
    }
}
```

* The non-static Variables will be always loaded with default values first & then the non-static initializers gets exeuted in top to bottom order.

* In the above example non-static variables "a" & "b" will be loaded inside the object with the default values & then non-static initializers a=10 & b=20 in top to bottom order.

Non-Static block:

* A block Created inside the class without static keyword is called "non-static block".
* A non-static block is a type of non-static initializer
* It is also Called as "Instance Initializer block" (IIB).

Syntax:
```
{
    statements;
}
```

Note: * Non static block doesn't have any name. Therefore We Cannot explicit name
* Non static block doesn't have return type. It Cannot return any value.
* Non static block get executed during object Creation.
* We Can have multiple non-static block inside the class & then executed in top to bottom order.

Non-Static block & Constructor :

* All the non static block of a class gets inserted inside the Constructor by the Compiler.
* The non-Static block gets inserted before the user defined statement of the Constructor.

ex: Before Compiling

```
class M
{
  { s.o.pln("from IIB-1");
  }
  M()
  {
    s.o.pln("from Constructor M()");
  }
  p.s.v.m(String[] args)
  {
    s.o.pln("from main()");
    M obj1 = new M();
  }
}
```

After Compiling.

```
class M
{
  M()
  {
    //non static block inserted by Compiler.
    {
      s.o.pln("from IIB-1");
    }
    s.o.pln("from Constru M()"); //user defined instruction of Constr.
  }
  p.s.v.m(S[] args)
  {
    s.o.pln("from main()");
    M obj1 = new M();
  }
}
```

```
class N
{
 int a;
 int b;
 {
   s.o.pln("from IIB-1 of N");
   a=10;
 }
 {
   s.o.pln("from IIB-2 of N");
   b=20;
 }
 N()
 {
   s.o.pln("from N()");
   s.o.pln(a);
   s.o.pln(b);
 }
 N(int i)
 {
   s.o.pln("from N(int)");
 }
 P.S.V.m(String[] args)
 {
   s.o.pln("from main()");
   N obj1 = new N();
   N obj2 = new N(10);
   s.o.pln(obj1.a);
   s.o.pln(obj1.b);
 }
}
```

o/p

```
From main()
From IIB-1 of N
From IIB-2 of N
From N()
10
20
From IIB-1 of N.
From IIB2-of N
From N(int)
10
20
```

Note: A variable declared inside a non-static block is a local variable of that block.

ex:

```
class A
{
  {
    int a; // Local Variable
  }
  P.S.V.m(S[] args)
  {
    A obj1 = new A();
    S.o.pln(obj1.a); // CTE
  }
}
```

Non-Static Method:

* A method without static keyword prefixed with it called non-static method.
* Non-static methods belongs to objects of the class therefore we must create object before calling / using non-static method.

ex:

```
class P
{
    public void test()  <- Non-static method
    {
        s.o.pln("from test()");
    }
    p.s.v.m(s[] args)
    {
        s.o.pln("from main()");
        P obj1 = new P();
        obj1.test();
    }
}
```

o/p

From main()
From test()

Note:- We can use non-static method of one class in another class by creating object of that class. as shown in below example.

```
class Q
{
    public void a1()
    {
        s.o.pln("from a1() of Q");
    }
}

class R
{
    p.s.v.m(s[] args)
    {
        Q obj1 = new Q();
        obj1.a1();
    }
}
```

Non-Static Context:

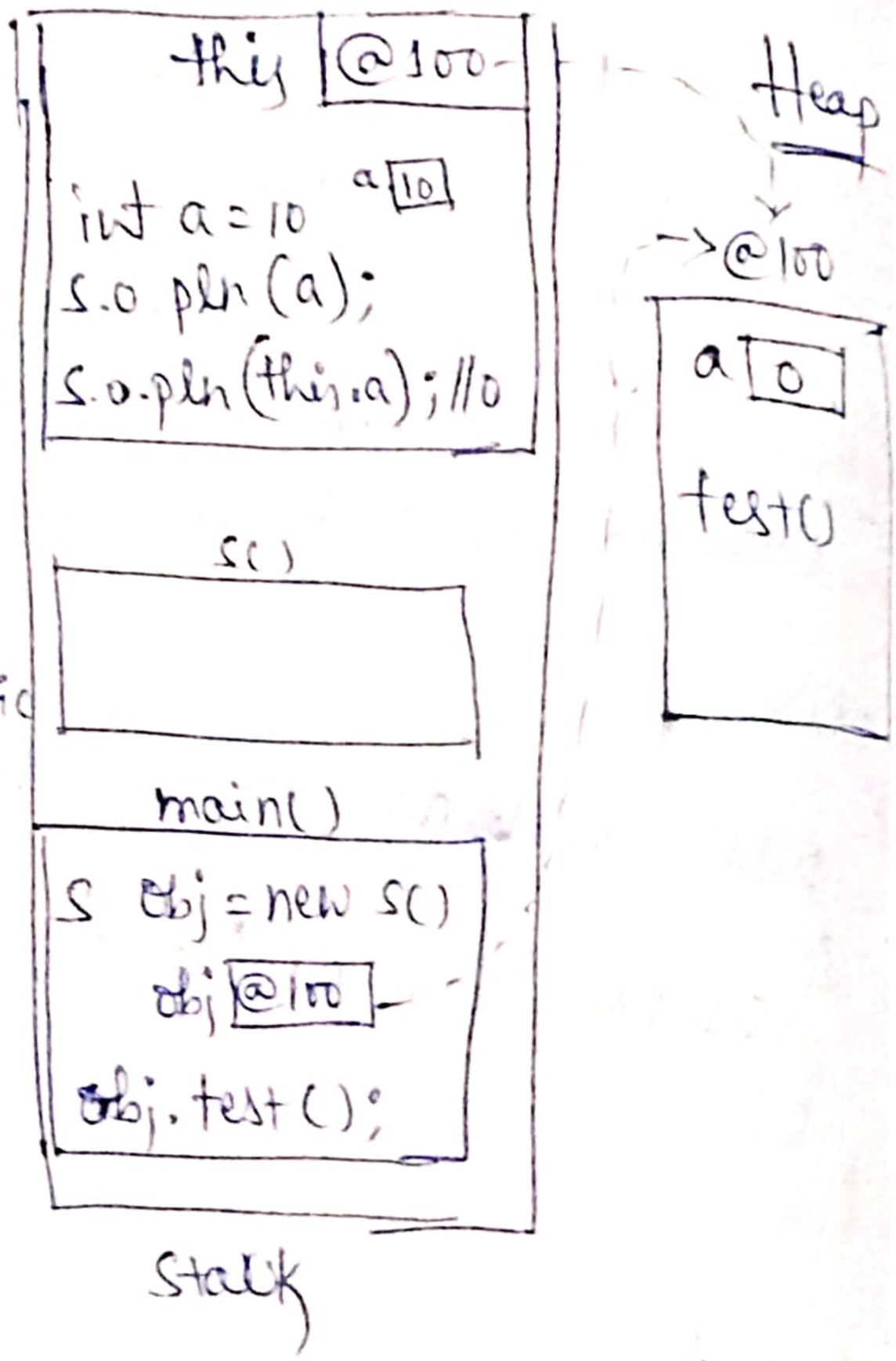
* Non-Static Context includes non-static block, non-static methods & Constructor.
* Inside each & every non-Static Context there will be "this". "this" holds the reference of the object to which non-Static Context belongs.

This:

* This is a Keyword in java which acts as pre-defined local object reference Variable it holds the address/reference of the Current object.

ex: class S

```
{ int a;
  public Void test()
  {
    int a=10;
    S.o.pln(a);//10 ← local Variable
    S.o.pln(this.a);//0 ← non-Static Variable
  }
  p.s.v.m(S[] args)
  {
    S obj=new S();
    obj.test();
  }
}
```



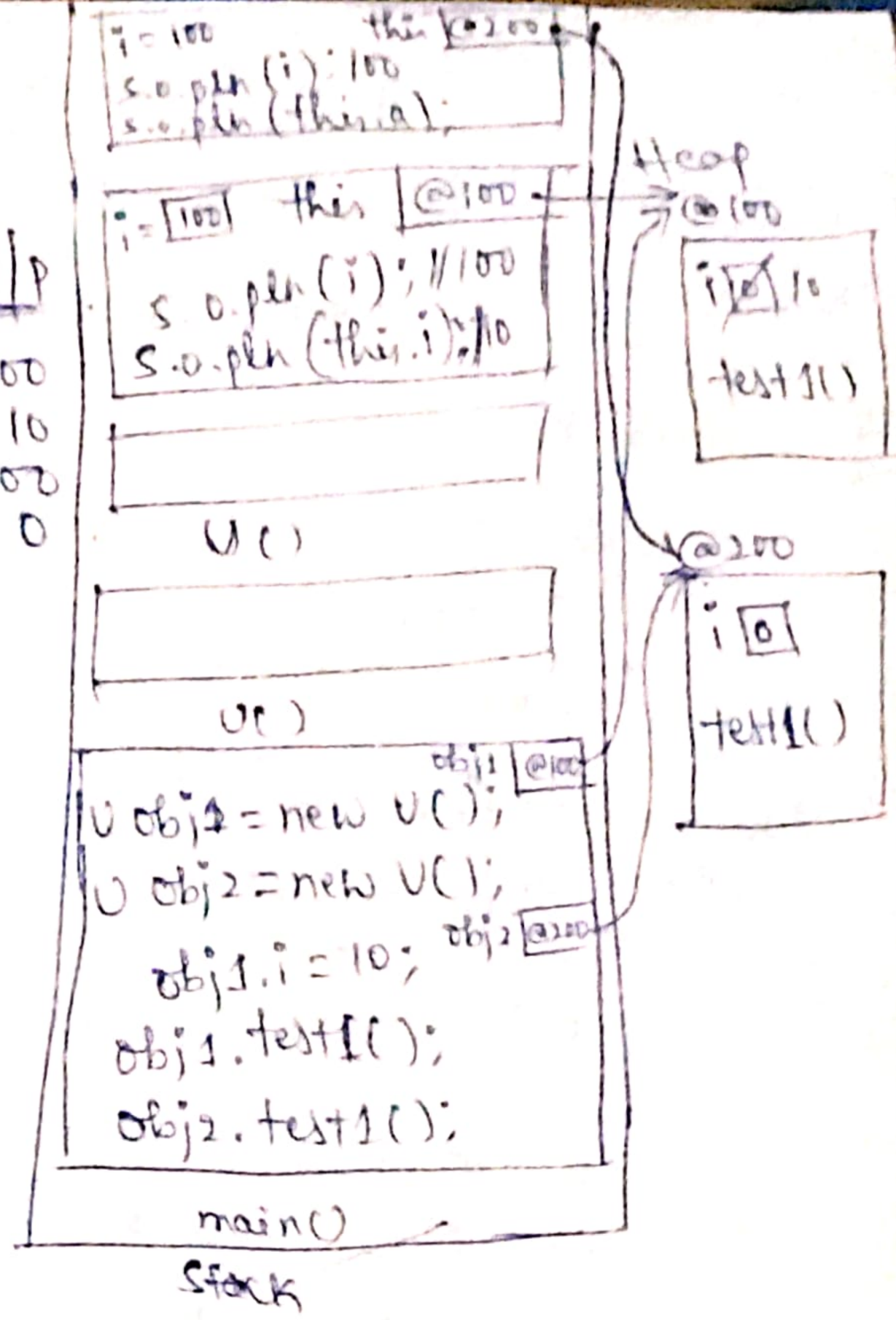
* Inside a non-Static Context if there is a local Variable having same name as non-Static Variable the highest Priority will be given to local Variable in this Case if we Want to access the non-Static Variable we should make use of "this" Keyword

ex: class T

```
{ int a;
  T(int a) ← Constructor
  {
    S.o.pln(a);//10 local Var
    S.o.pln(this.a);//0 non Static Var.
  }
  p.s.v.m(S[] args)
  {
    T obj1=new T(10);
  }
}
```

ex:
```
class U
{ int i;//10
  public Void test1()
  {
    int i=100;
    s.o.pln(i);
    s.o.pln(this.i);
  }
  p.s.v.m(S[] args)
  {
    U obj1=new U();
    U obj2=new U();
    obj.1=10;
    obj1.test();
    obj2.test();
  }
}
```

o/p
100
10
100
0



Note: "This" Keyword Cannot be used from a Static Context (static method (or) static block).

ex:
```
class V
{ int a;
  p.s.v.m(string[] args) <= static context
  {
    V obj1=new V();
    s.o.pln(this.a);//CTE
  }
}
```

Purpose of Using Parameterized Constructor

* With help of Parameterized Constructor we can assign diff values for non-static variables in diff objects.

* ex

```
class W
{
    int a;
    int b;
    W(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
    p.s.v.m(S[] args)
    {
        W obj1 = new W(10,20);
        W obj2 = new W(30,40);
        W obj3 = new W(50,60);
        S.o.pln(obj1.a);//10
        S.o.pln(obj1.b);//20

        S.o.pln(obj2.a);//30
        S.o.pln(obj2.b);//40

        S.o.pln(obj3.a);//50
        S.o.pln(obj3.b);//60
    }
}
```

Non static variable — this.a — local variable

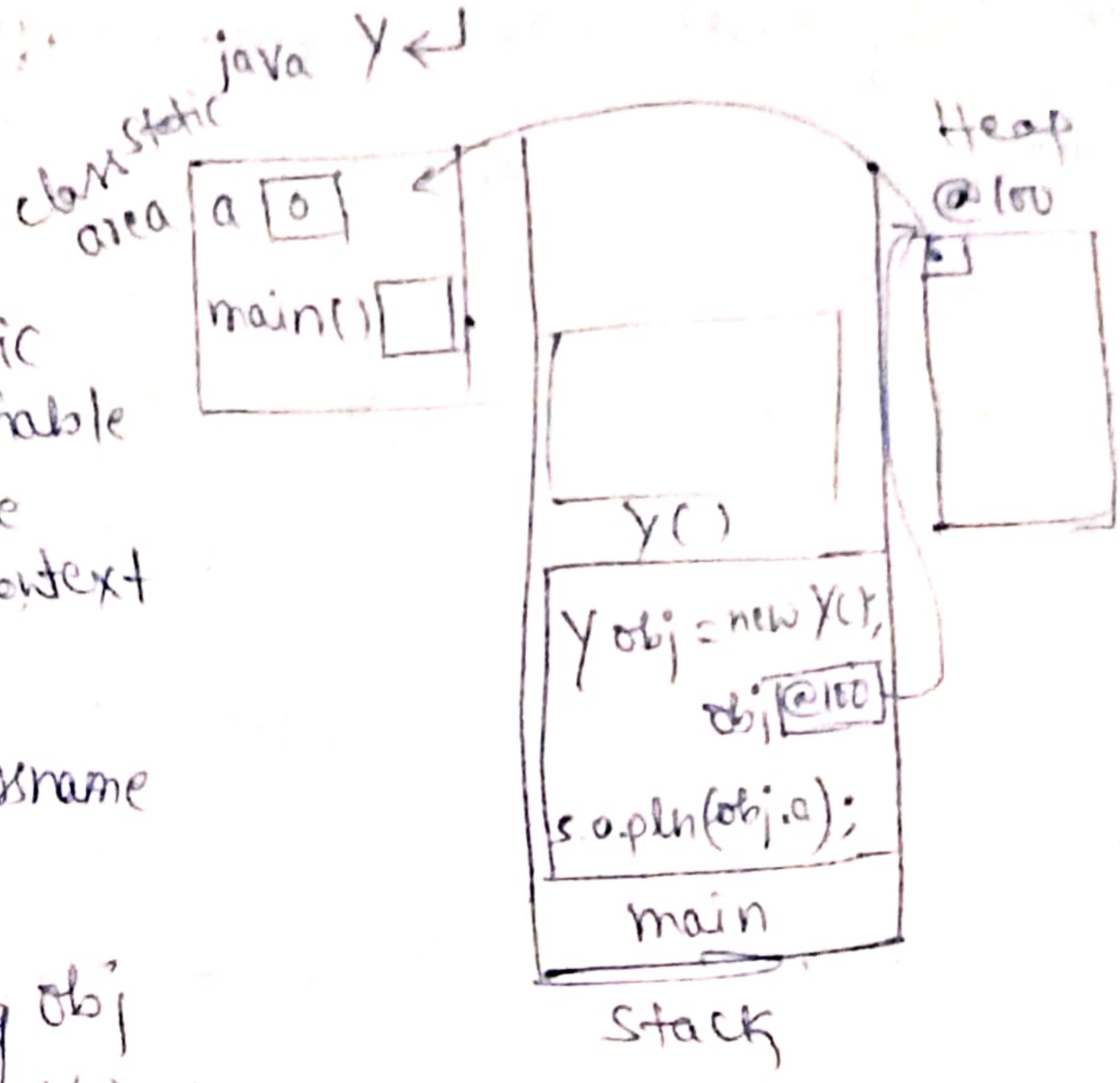
Using Static & Non-Static Members from Static & Non-Static Context

(1) From Static Context :

* (a) Using Static members from Static Context :-
  - (i) Directly
  - (ii) Using classname
  - (iii) Using object reference.

ex

(iii) Using object reference :-




```
ex: class Y
{
    static int a; // Static Variable

    p.s.v.m(S[] a) ← static Context
    {
        s.o.pln(a); // Directly.
        s.o.pln(Y.a); // using Classname
        Y obj=new Y();
        s.o.pln(obj.a); // using obj reference Variable.
    }
```

(b) Using non-static members from static Context :

(i) Using object reference.

```
ex: class Y
{
    int a; // non-static Variable

    p.s.v.m(S[] args) ← static Context i.e static method.
    {
        Y obj = new Y();
        s.o.pln(obj.a); // Using object reference
```

(2) From Non-static Context :-

(a) Using non-static members from non-static Context :

(i) Directly

(ii) Using "this" Keyword.

```
ex: class Y
{
    int a; // Non-static Variable

    Y() ← Non Static Context i.e Contructor
    {
        s.o.pln(a); // directly
        s.o.pln(this.a); // Using this keyword.
    }

    p.s.v.m()
    {
        Y obj = new Y();
    }
```

(b) Using static members from non static Context :

(i) Directly
(ii) Using this
(iii) Using classname.

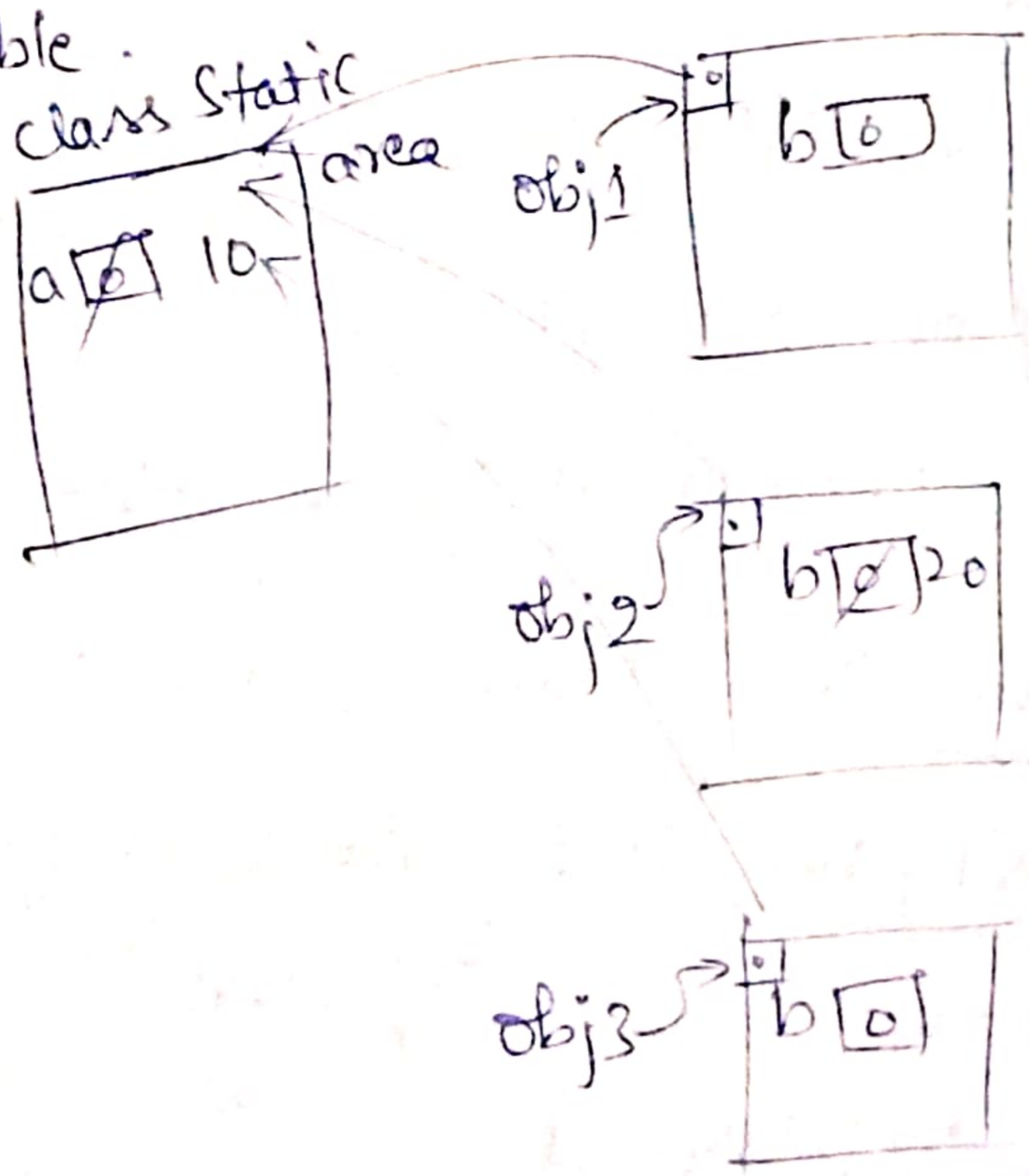
ex: class Y
{
static int a; // static Variable
Y() ⇐ Non-Static Context
{
s.o.pln(a); // directly
s.o.pln(this.a); // using this Keyword
s.o.pln(Y.a); // Using classname
}
p s.v.m()
{
Y obj = new Y();
}
}

Note: Inside a class We Can't have static & non-static Variables With same name.

ex: class Z
{
static int a; // static Variable
int b; // Non-Static Variable.
p.s.v.m()
{
Z obj1 = new Z();
s.o.pln(obj1.a); //0
s.o.pln(obj1.b); //0

Z obj2 = new Z();
obj2.a = 10;
obj2.b = 20;
s.o.pln(obj2.a); //10
s.o.pln(obj2.b); //20

Z obj3 = new Z();
s.o.pln(obj3.a); //10
s.o.pln(obj3.b); //0.
}
}



o/p
0
0
10
20
10
0

```
ex: class N1
{   int a;
    int b;
    {
        s.o.pln("from IIB-1 of N1");
        int a=10;
        s.o.pln(this.a);
        s.o.pln(b);
        b=10;
    }
    {
        s.o.pln("from IIB2 of N1");
        s.o.pln(a);
        s.o.pln(b);
    }
    N1(int a, int b)
    {   s.o.pln("from Constructor N1(int,int)");
        this.a=a;
        this.b=b;
    }
}
class N2
{   int c;
    {   s.o.pln("from IIB-1 of N2");
    }
    N2()
    {   s.o.pln("from Constructor N2()");
        s.o.pln(c);
    }
    N2(int c)
    {   s.o.pln("from Constructor N2(int)");
        s.o.pln(c);
    }
    public Void test()
    {   s.o.pln("from test");
        c=50;
        s.o.pln(this.c);
    }
    p.s.v.m(String[] args)
    {   s.o.pln("from main() of N2");
        N1 obj1 = new N1(30,40);
        s.o.pln(obj1.a);
        s.o.pln(obj1.b);
        N2 obj2 = new N2();
        obj2.test();
        N2 obj3 = new N2(20);
```

Java N2 ↵

o/p

```
from main() of N2
from IIB-1 of N1
0
0
from IIB-2 of N1
0
10
from Constru N1(int,int)
30
40
from IIB-1 of N2
from Contr N2()
0
from test
50
from IIB1 of N2
from Contr N2()
20.
```

ex: class S1

```
{
    static int a;
    int i;
    static
    {
        s.o.pln("from SIB-1 of S1");
    }
    {
        s.o.pln("from IIB-1 of S1");
        s.o.pln(a);
    }  s.o.pln(i);
    public static int test1()
    {
        s.o.pln("from test1()");
        a=5;
        return 10;
    }
    public int test2()
    {
        s.o.pln("from test2()");
        return 20;
    }
    P.S.V.m()
    {
        s.o.pln("from main() of S1");
        int a=test1();
        s.o.pln(S1.a);
    }
}// class S2
{
    int b;
    static
    {
        s.o.pln("from SIB-1 of S2");
    }
    {
        s.o.pln("from IIB-1 of S2");
    }
    S2(int b)
    {
        s.o.pln("from Constructor S2(int)");
        this.b=b;
    }
    P.S.V.m()
    {
        s.o.pln("from main() of S2");
        S1 obj1 = new S1();
        int b = obj1.test2();
        S2 obj2 = new S2(b);
        s.o.pln(obj2.b);
        S1.main(null);
    }
}
```

Java S2

o/p

~~from SIB-1 of S2~~

from main of S2

from SIB-1 of S1

from IIB-1 of S1

0

0

from test2()

from IIB-1 of S2

from Contr S2(int)

20

from main of S1

from test1()

5

Object Oriented Programming (OOP's)

Object Oriented Programming is a theory/Concept/technique which helps the programmer to represent real world scenarios through programming

Important Terminologies under OOP's

* Class
* Object
* Encapsulation
* Inheritance
* Polymorphism
* Abstraction.

Object:

* Object is a real world entity which has properties & behaviours.

Properties are also called as attributes/fields.

Behaviours are also called as Actions/tasks.

* In java, Properties of an object is represented as "Variables" (non-static)
* Behaviours are represented as "methods" (Non-static).
* Since java supports Object Oriented Programming it is called as "Object Oriented Programming lang".

Class:

Class is a blueprint of an object which represents properties & behaviours of the object as variables & methods respectively.

OR

Class is a template of an object which represents properties as variables & behaviours as methods

Pillers of Object Oriented Programming

Encapsulation:

Process of binding Properties & behaviours of an Object together.

Inheritance:

Process of acquiring Properties & behaviours ~~Property~~ of Parent to child.

Polymorphism:

Ability of an object to exhibit more than One form

Abstraction:

Process of hiding the implementation details

Encapsulation:

* It is the process of binding properties & behaviours of an object together.

ex:

Employee

Properties
* Name
* Eid
* Sal
* designation

Behaviour
Work.

- String Name;
  Int eid;
  double sal;
  String designation;

* public Void Work()
  {
  }

Example for Encapsulation Without data hiding

ex:
```
class Employee
{
    String name;
    int eid;
    double Salary;
    String designation;
  Employee().
  {
  }
  Employee (String name, int eid, double Salary, String designation)
  {
      this.name = name;
      this.eid = eid;
      this.Salary = Salary;
      this.designation = designation;
  }
  public Void Work()
  {
      S.o.pln(name + " is Working as " + designation);
  }
}// end of employee
```

Parametrised Constructor

```
Class Driver
{
    P s v m(SE7 args)
    {
        Employee e1 = new Employee("Raj", 1, 50000.0, "TE");
        displayDetails(e1);

        Employee e2 = new Employee("Rahul", 2, 50000.0, "Dev");
        displayDetails(e2);

        e1.salary = 60000.0; //modifying value of Variable
        displayDetails(e1);
    }
    public Static Void displayDetails(Employee e)
    {
        S.o.pln("Name:" + e.name); //reading value of Variable
        S.o.pln("Eid:" + e.eid);
        S.o.pln("Salary:" + e.salary);
        S.o.pln("Designation:" + e.designation);
        e.Work(); // Calling Work method.
    }
}
```

* In above example we can read & modify the values of the Variables of Employee class from another class.

* The standard defin^n of encapsulation says "the direct access for reading & modifying the value of the Variable Outside the class should not be given.

* Standard definition for Encapsulation :-

The process of hiding internal states & requiring all the interactions to be performed only through non-static methods is called "Encapsulation".

Internal states means ⇒ Properties/Variables of the class

Interaction means ⇒ Reading & modifying value of the Variable.

Data Hiding

* The process of hiding the properties from the outside world is known as data hiding.
* We can achieve data hiding with the help of Private access Specifier

Private

* It is a keyword which acts as access specifier, when a member of a class is made private, it can be used only inside the same class. We cannot use it outside the class

Note: In order to use a private variable of a class outside the class, we should make use of "getter" & "Setter" methods

"Getter" → method is used for reading the values of a variables.

"Setter" → method is used for modifying the value of a variables

* If the private variable should be both readable as well modifiable outside the class we should design both getter & setter method.

* If private variable should be only readable outside the class then we should design only getter method.
* If private variable should be only ~~readable~~ modifiable outside the class then we should design only setter method.

Syntax of Getter Method

```
public datatype_of_the_attribute getName_of_Attribute()
{
    return Attribute;
}
```

Ex:

```
Public String getName()
{
    return name;
}
```

```
public String getDesignation()
{
    return designation;
}

public void setDesignation(String designation)
{
    this.designation = designation;
}
}

class Employee
{
    private String name, String designation;
    private double salary;
    private int id;
    public Employee(int id, String name, double salary, String designation)
    {
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.designation = designation;
    }
    public void show()
    {
        System.out.println("Name: " + e.getName());
        System.out.println("ID: " + e.getID());
        System.out.println("Salary: " + e.getSalary());
        System.out.println("Designation: " + e.getDesignation());
    }
}

class EmployeeTest
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee(101, "Ram", 15000, "Manager");
        displayDetails(e1);
        Employee e2 = new Employee(102, "Sabal", 8000, "Dev");
        displayDetails(e2);
        e1.setSalary(60000);
        displayDetails(e1);
    }
    public static void displayDetails(Employee e)
    {
        System.out.println("Name: " + e.getName());
        System.out.println("ID: " + e.getID());
        System.out.println("Salary: " + e.getSalary());
        System.out.println("Designation: " + e.getDesignation());
    }
}
```

Syntax of Setter Method

```
public Void SetName_of_Attribute(datatype_of_attribute) attribute
{
    this.attribute = attribute;
}
```

ex

```
public Void SetName(String name)
{
    this.name = name;
}
```

ex:

```
Class Employee
{
    private String name;
    private int eid;
    private double Salary;
    private String designation;
public String getName()
{
    return name;
}
public Void setName(String name)
{
    this.name = name;
}
public int getEid()
{
    return eid;
}
public Void setEid(int eid)
{
    this.eid = eid;
}
public double getSalary()
{
    return Salary;
}
public Void setSalary(double Salary)
{
    this.Salary = Salary;
}
```

// first letter Capital after get.

// Same name of non static Variable

```
public String getDesignation()
{
    return Designation;
}
public Void setDesignation(String designation)
{
    this.designation = designation;
}
Employee()
{
}

Employee(String name, int eid, double salary, String designation)
{
    this.name = name;
    this.eid = eid;
    this.Salary = salary;
    this.designation = designation;
}
public Void Work()
{
    s.o.pln(name+" is Working in"+designation);
}
}// end of Employee Class

class Driver1
{
    p.s.v.m(S[] args)
{
    Employee e1 = new Employee("Raj", 1, 50000.0, "TE");
    displayDetails(e1);

    Employee e2 = new Employee("Rahul", 2, 50000.0, "Dev");
    displayDetails(e2);

    e1.setSalary(60000.0);
    displayDetails(e1);
}
    public Static Void displayDetails(Employee e)
{
    s.o.pln("Name:"+e.getName());
    s.o.pln("Eid:"+e.getEid());
    s.o.pln("Salary:"+e.getSalary());
    s.o.pln("Designation:"+e.getDesignation());
    e.Work();
}
```

Advantages of Encapsulation

* With the help of encapsulation we can achieve data hiding which provides security to the data. also we can perform some verification & validation. With the help of getter & setter methods..

ex:

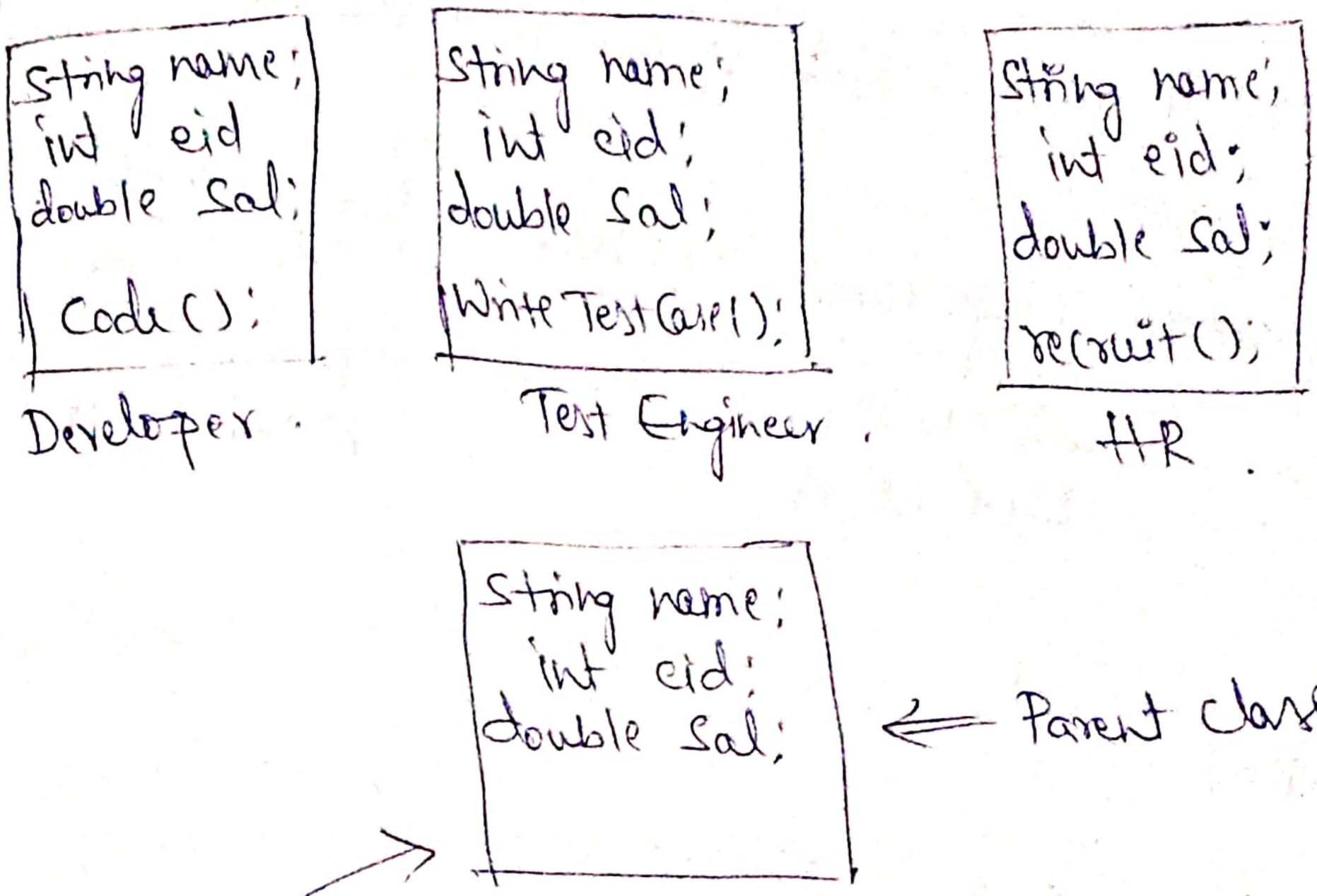
```
public void setSalary(double salary)
{
    if (salary > this.salary)
    {
        this.salary = salary;
        S.o.pln("Salary is updated");
    }
    else
    {
        S.o.pln("Sorry! Salary can't be updated");
        S.o.pln("New salary must be greater than
                current salary");
    }
}
```

Inheritance :-

The process of Child class aquiring properties & behaviours from it's parent class is called "Inheritance".

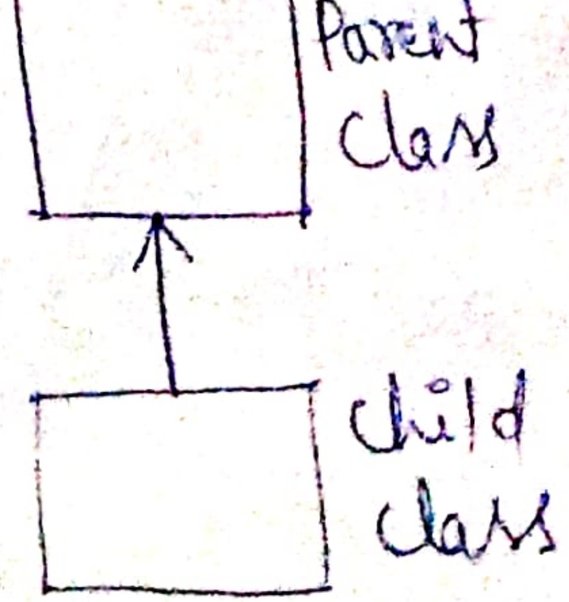
* Inheritance is also called as "is-a-relationship".

ex: Developer is an Employee ← Parent class
Car is a Vehicle ←



* In the above example employee class acts as Parent class for developer, Test Engineer, HR classes therefore all the child classes i.e. developer, TE, HR classes can aquire the properties & behaviours from employee class.

* Inheritance b/w two classes can be represented with the help of an arrow as shown in below diagram & arrow head should always point towards Parent.



* The parent class is also called as "Base class / Super class".
* The child class is also called as "Derived / Sub Class".
* The process of sub class acquiring properties & behaviour from it's super class is called "inheritance".
* Major advantage of inheritance is "Code re-usability".

Extends :-

* is a keyword in java which is use to achieve inheritance b/w two classes.

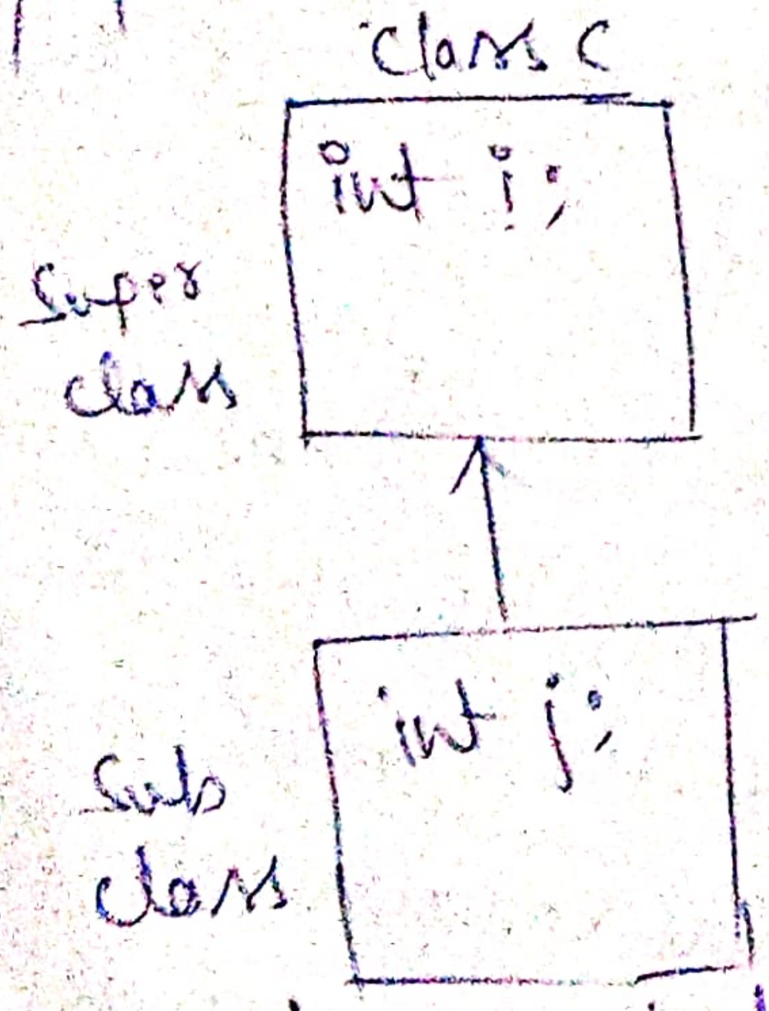
Syntax: class class_name extends class_name

class_name → sub class / child class

class_name → super class / parent class

```
class A
{
  static int a;
}

class B extends A
{ p.s.v.m
  s.o.pln(a); //0
}
```

ex.



i ⇒ inherited Member

j ⇒ incorporated member.

* In above example class D extends C therefor D becomes sub class for class C. & sub class D will have inherited members from super class C as well as its own members (incorporated members).

Inheritance with respect to Static Members

* With respect to inheritance during class loading process super class will be always loaded first.
* Sub classes are loaded immediately after super class.
* the sub class static area will have reference to its super class static area.

ex:
```
class C
{
    static int i=10;
    public static void test()
    {
        s.o.pln("from test() of C");
    }
}

class D extends C
{
    static int j=20;
    public static void main(String[] args)
    {
        s.o.pln(j);
        s.o.pln(D.j);
        D obj=new D();
        s.o.pln(obj.j);

        s.o.pln(i);//Directly
        s.o.pln(D.i);//Using subclass name
        s.o.pln(obj.i);//using subclass object

        test();
        D.test();
        obj.test();
    }
}
```

* The static member of super class can be used in subclass
  (i) Directly
  (ii) Using subclass name
  (iii) Using subclass object reference.

Inheritance With respect to Non-Static

* when we create an obj of sub class the sub class object will have all the non-static members of super class as well as same class loaded inside it.
* therefore with help of sub class type object reference variable we can access non-static members of super class as well as same class.

ex:
```
class E
{
  int i;
  public void test()
  {
    s.o.pln("from test() of E");
  }
}
class F extends E
{
  int j;
  public void test1()
  {
    s.o.pln("from test1()");
  }
}
class Driver1
{
  public static void main(String[] args)
  {
    E obj1 = new E();
    s.o.pln(obj1.i); //0
    obj1.test(); // from test() of E

    s.o.pln(obj1.j); //CTE
    obj1.test1(); //CTE.

    F obj2 = new F();
    s.o.pln(obj2.j); //0
    obj2.test1(); // from test1()

    s.o.pln(obj2.i); //0
    obj2.test(); // from test of E
  }
}
```

ex: class G
{
static int a;
int i;
}
class H extends G
{
static int b;
int j;
}
Class Driver2
{
p.s.v.m()
{
G obj1 = new G(); // creating object for Super class.
S.o.pln(obj1.a); // Accessing Superclass Variable using Super class Obj.
S.o.pln(obj1.i); // Accessing Super class non-Static Variable using obj of Super cl
S.o.pln(obj1.b); // Access Sub class static Variable using obj of Super class
S.o.pln(obj1.j); // Accessing Sub class non-static Var using obj of Super class

H obj2 = new H(); // Creating obj of Subclass.
S.o.pln(obj2.b); // Accessing Subclass static Var. using obj of Subclass.
S.o.pln(obj2.j); // Accessing Subclass non-static Var. using obj of Subclass.
S.o.pln(obj2.a); // Access Super class static Var using obj of Sub class.
S.o.pln(obj2.i); // Accessing Super class non-static Var. using obj of Sub class
}

	Super class		Sub class	
	Static members	Non-static members	Static members	Non-static Members
Using Super class type object ref Variable	✓	✓	✗	✗
Using Subclass type object ref Variable	✓	✓	✓	✓

Constructor Chaining :

* It a process of one Constructor Calling another Constructor.
* So we Can achieve Constructor Chaining in 2 ways.

(i) Using Super() Statement.
(ii) Using this() Statement.

(i) Using Super() Statement :

* Super() Statement is used to Calling Constructor of Super Class.
* Super() Statement must be used only inside the Constructor.
* Super() Statement must be the first Statement of the Constructor
* If programmer fails to add Super() Statement inside the Constructor then Compiler by default adds a Super() Statement without any arguments.

If Super class has multiple Constructors which Constructor of the Super class will be Called by Super() Statement ?

* It depends on the arguments passed in the Super Call Statement.

ex Super() ⇒ it Calls No argument Constructor of Super class

Super(10) ⇒ It Calls Constructor of Super class which has integer argument.

ex:-
```
class A
{
    static int a=10;
    int i=20;
    A()
    {
        S.o.pln("from Constructor A()");
    }
}
```

```
class B extends A
{ static int b=30;
  int j=40;

  B()
  { s.o.println("from Constructor B()");
  }
}
class C extends B
{ static int c=50;
  int k=60;
  C()
  { s.o.println("from Constructor C()");
  }
}
class Driver1
{ p.s.v.m(s[] args)
  {
    C obj1 = new C();
  }
}
```

* In the above example Compiler adds super() statement without any arguments inside the Constructors of Class A, B, C.

Note: In java for a class if extends Keyword is not used it by default extends pre-defined class called "object class".

* "Object class is the super most class for all the classes in java

Order of Constructor invocation ⇒ C() → B() → A() → object()

Order of Completion of execution ⇒ object() → A() → B() → C()

ex:
```
class A
{
    A()
    {
    }
    A(int a)
    {
    }
    A(int a, double a)
    {
    }
}.
class B extends A
{
    B(char a)
    {
        super();
    }
    B(int a, int b)
    {
        super(10,20);
    }
}
class C extends B.
{
    C(double a)
    {
        super('a');
    }
    C(int a)
    {
        super(5, 6.5);
    }
}.
```

```
class Driver1
{
  P.S.V.m(S[] args)
  {
    new A();
    new A(10,20);
    new B('a');
    new B(1,2);
    new C('a');//CTE
    new C();//CTE
  }
}
```

Order of Constructor invocation ⇒

Order of Completion Execution ⇒

(ii) Using this() Statement :-

* this() Statement is used to call the Constructor of Same class.
* this() statement must be used only inside the Constructor.
* this() Statement must be the first Statement of the Constructor (Invid a Constructor this() Statement & super() Statement Can't be used together).
* For the Constructor having this() Statement Compiler will not add any instructions.
* We Cannot Call Constructor recursively using this call Statement it generates Compile time error.

ex
```
class D
{
  D()
  {
    this(10);//Recursive//CTE
    S.o.pln("from D()");
  }
  D(int a)
  {
    this();//Recursive//CTE
    S.o.pln("from D(int)");
  }
  p.s.v m()
  {
    D.obj=new D();
  }
```

* Before Compiling

```
class E
{
    {
      s.o.pln("from IIB-1 of E");
    }
    E()
    {
      S.O.pln("From E()");
    }
    E(int a)
    {
      this(0);
      S.O.pln("from E(int)");
    }
    p.s.v m()
    {
      E obj1=new E();
      E obj2=new E(10);
    }
}
```

o/p

```
obj1 { from IIB-1 of E
       From E()
obj2 { from IIB-1 of E
       from E()
       from E(int)
```

*

```
class E
{
    E()
    {
      super();
      {
        s.o.pln("from IIB 1 of E");
      }
      s.o.pln("From E()");
    }
    E(int a)
    {
      this(0);
      s.o.pln("from E(int)");
    }
    p.s.v.m()
    {
      E obj1 = new E();
      E obj2 = new E(10);
```

Q:

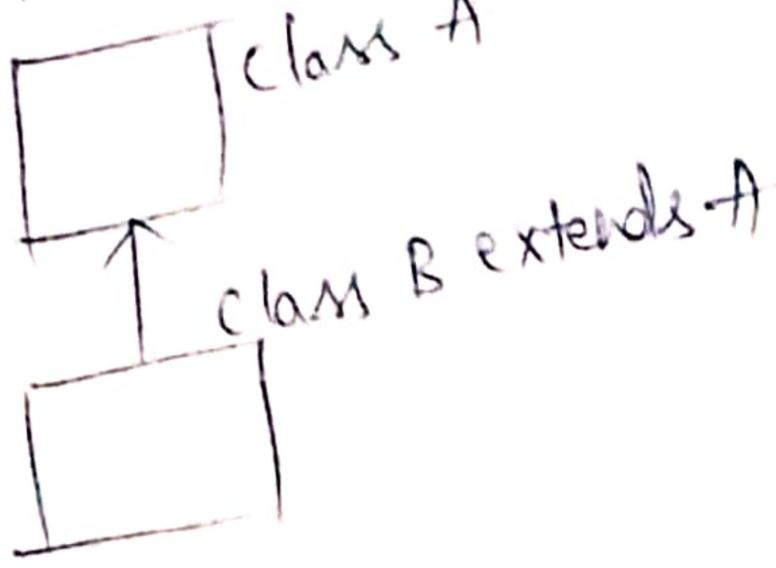
Types of Inheritance

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

(i) Single Level Inheritance :

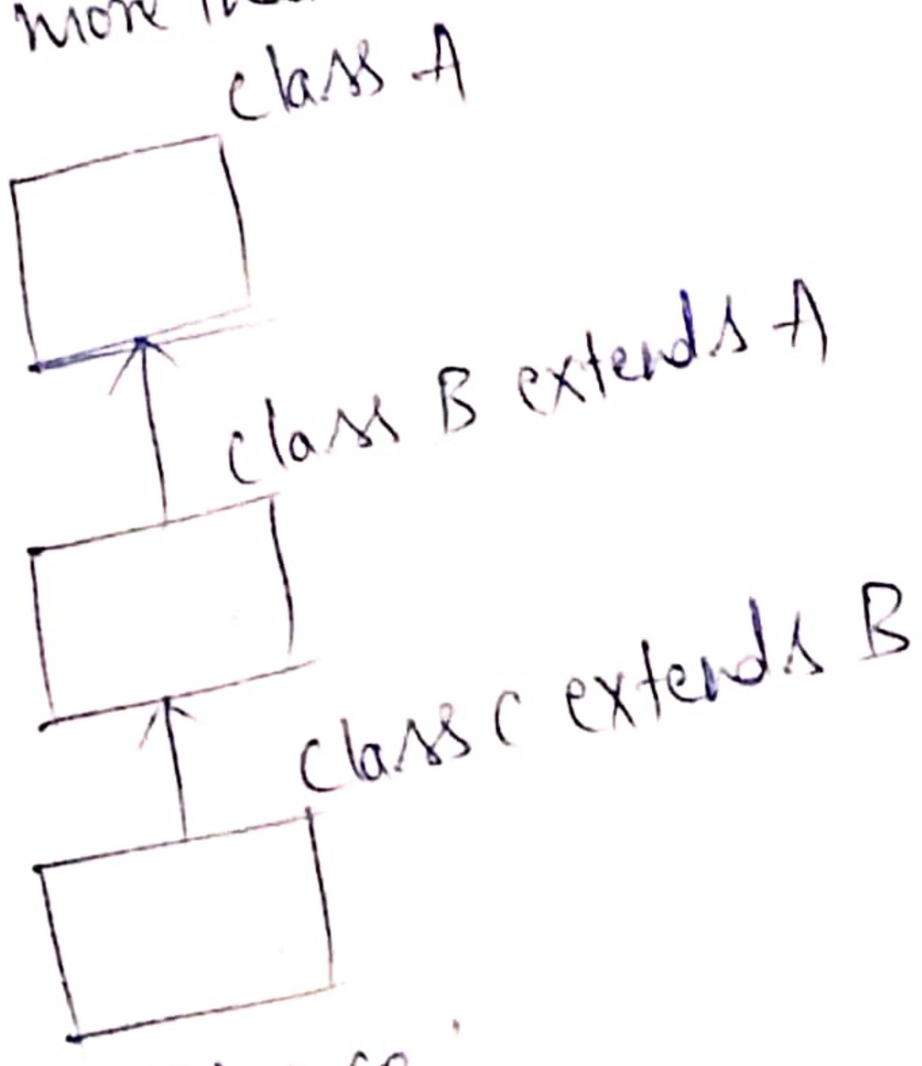
* Inheritance b/w only two classes is called single level inh

ex :



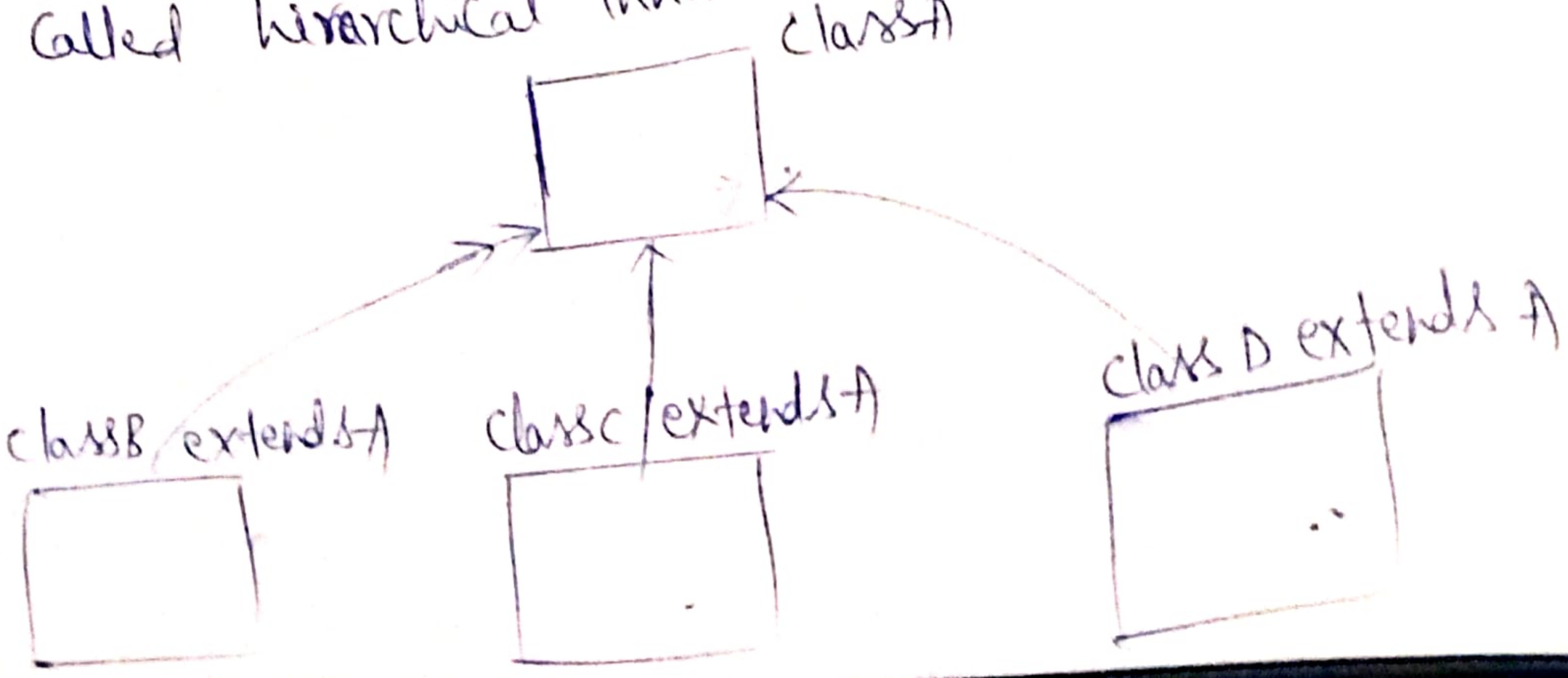
(ii) Multi Level Inheritance :

* Inheritance of more than one level is called multi level inheri



(iii) Hierarchical Inheritance :

* A super class having more than one subclass in the same level is called hierarchical inheritance



ex:
```
class A
{ A()
  { s.o.pln("from A()");
  }
}
class B extends A
{ B()
  { s.o.pln("from B()");
  }
}
class C extends A
{ C()
  { s.o.pln("from C()");
  }
}
class Driver1
{ p.s.v.m()
  { C obj1 = new C();
  }
```

o/p

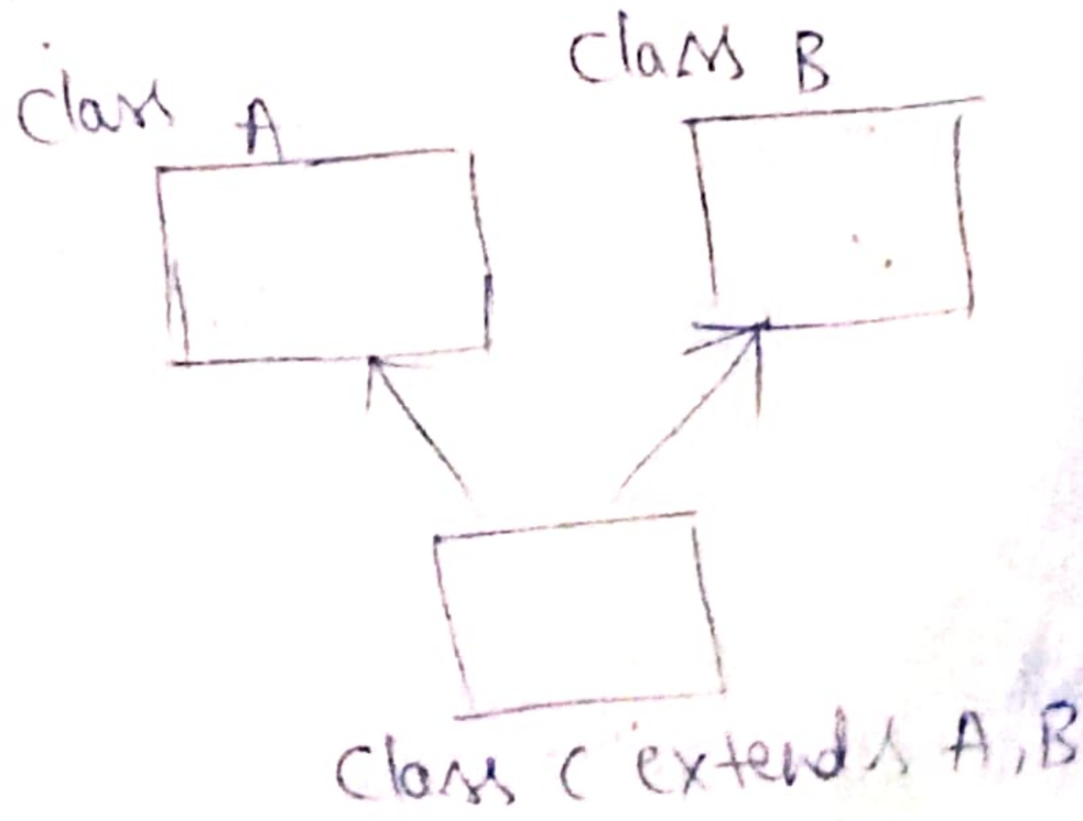
```
From A()
From C()
```

(iv) Multiple Inheritance:

A sub class derived from more than one super class in the same level is called

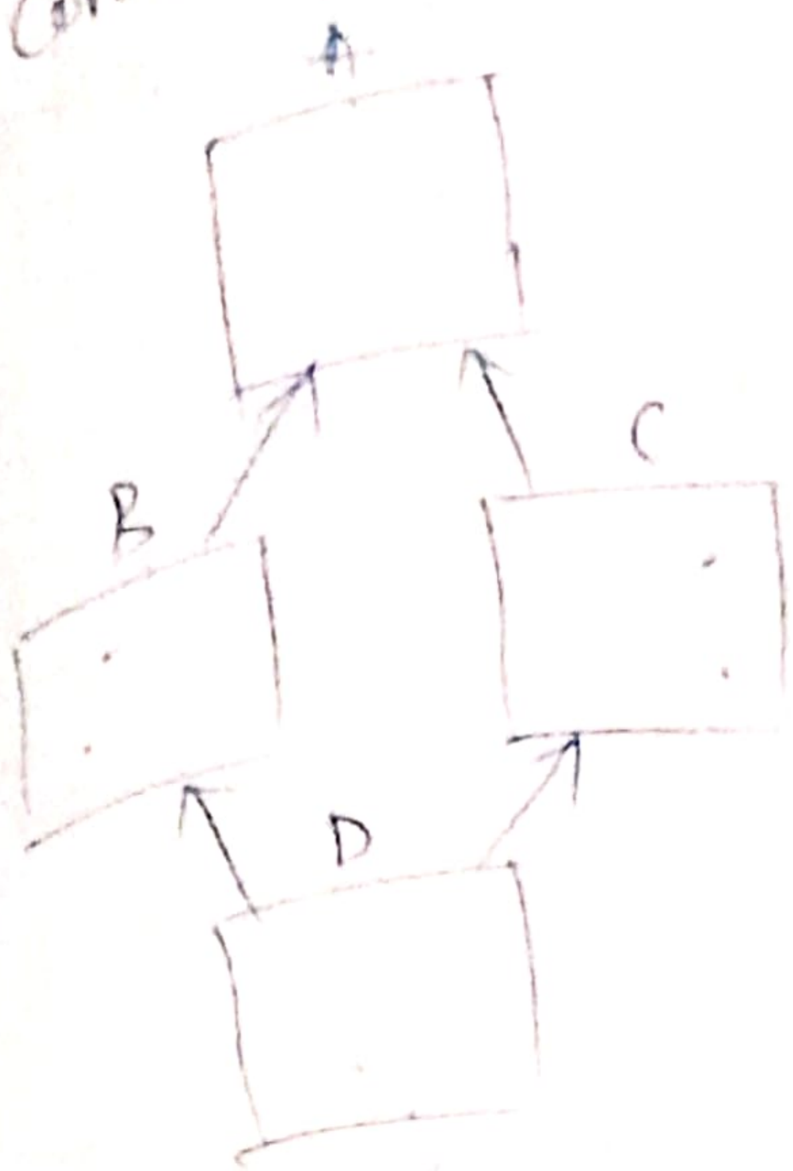
ex:
```
class A
{ static int a;
  int b;
}
class B
{ static int c;
  int d;
}
class C extends A,B
{
  static int e;
  int f;
}
```

```
class Driver2
{
  p.s.v.m()
  {
    C obj = new C();
  }
}
```



v) Hybrid inheritance

Combination of hierarchical & Multiple inheritance



* We Cannot achieve multiple inheritance using classes in java, Since hybrid inheritance involves multiple inheritance we Cannot achieve hybrid inheritance also.

ex:
```
class A
{
  int i;
}
class B extends A
{
  int j;
}
class C extends B
{
  int K;
}
```

```
class Driver1
{
  p.s.v m( )
  {
    A obj1 = new A();
    S.o.pln(obj1.i); //
    S.o.pln(obj1.j); //CTE
    S.o.pln(obj1.K); //CTE
  }
}
```

In the above example obj1 is object reference Variable A type (Super Class type) j & K are members of class B & C respectively (Sub class members).

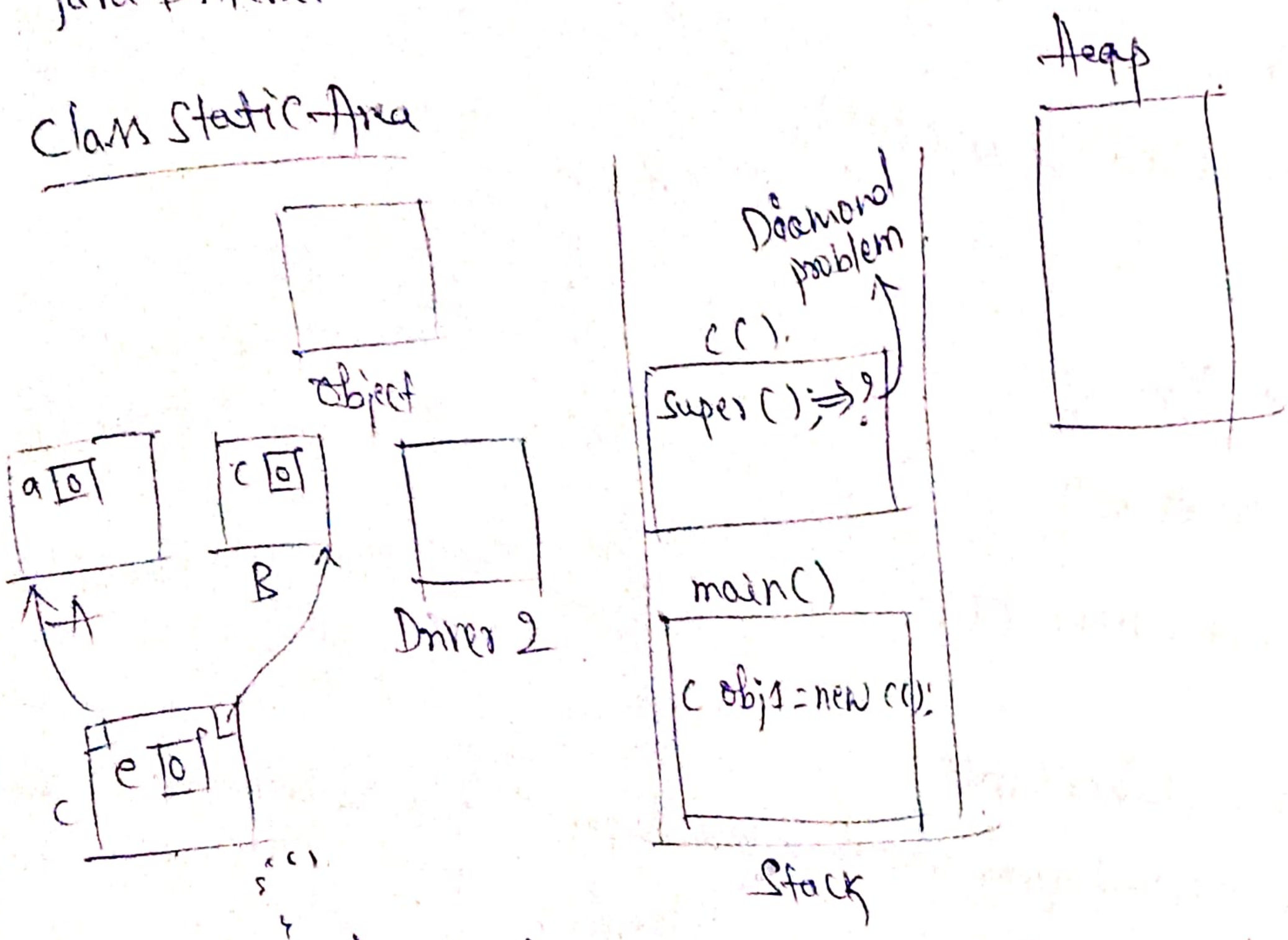
∴ using Super Class type object reference Variable obj1 we Cannot access Sub-class members j & K.

* Diamond Problem/Ambiguity :-

* In the above example class C extends class A & B therefore it has two super classes in the same level when we created an object of class C, constructor of class C is called & in the constructor of class C the first statement will be super() statement which should call constructor of super class.

* But for class C there are two super classes ~~that~~ now that arises an ambiguity in calling the constructor of super class & this ambiguity is called diamond problem/ambiguity.

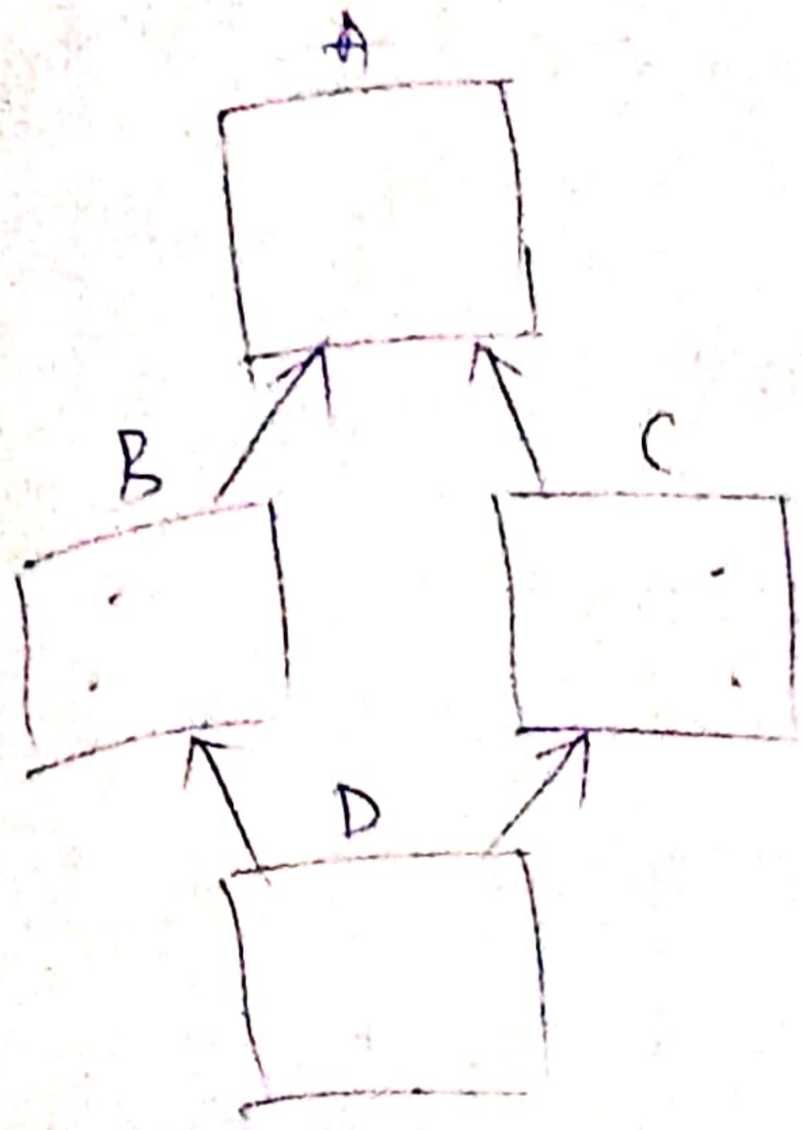
Java Driver2



* Because of diamond problem a class cannot extend more than one class. therefore we cannot achieve multiple inheritance using classes.

(v) Hybrid Inheritance

Combination of Hierarchical & Multiple inheritance



* We Cannot achieve multiple inheritance using classes in java, Since hybrid inheritance involves multiple inheritance we cannot achieve hybrid inheritance also.

Ex:

```
class A
{
  int i;
}
class B extends A
{
  int j;
}
class C extends B
{
  int K;
}
```

```
class Driver1
{
  P.S.V.M( )
  {
    A obj1 = new A();
    S.O.pln(obj1.i); //
    S.O.pln(obj1.j); //CTE
    S.O.pln(obj1.K); //CTE
  }
}
```

In the above example obj1 is object reference Variable A type (Super Class type) j & K are members of class B & C respectively (Sub class members).

∴ using Super Class type object reference Variable obj1 we Cannot access Sub class members j & K.

Non-Primitive Type Casting (Derived type Casting):-

* The process of Converting one type of object reference into another type of object reference is called non-primitive type Casting.

* 2 types of Non-Primitive type Casting.

(i) UpCasting (implicit)

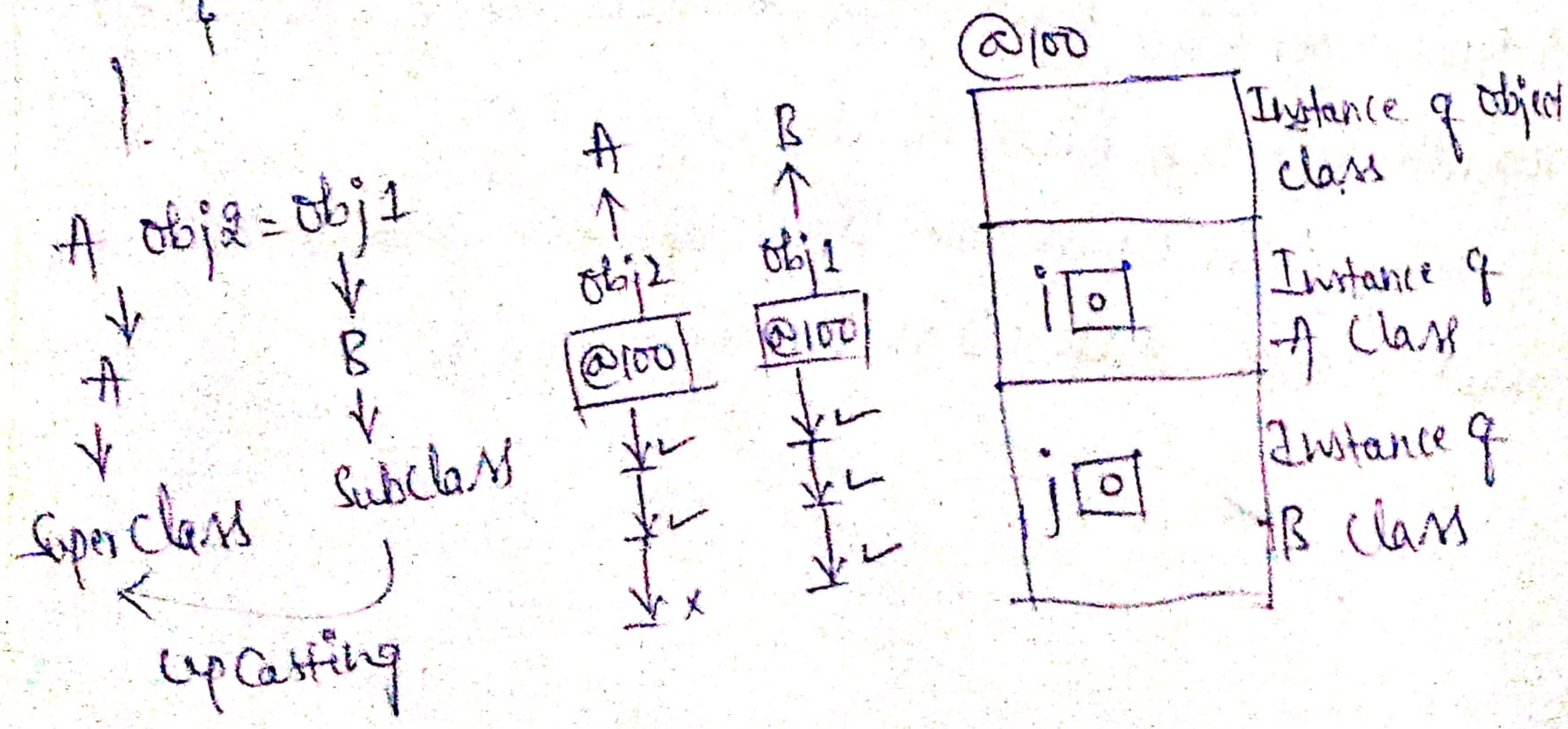
(ii) Down Casting (Explicit)

(i) Upcasting:

* The process of Converting Sub class type object reference into super class type is called upcasting.

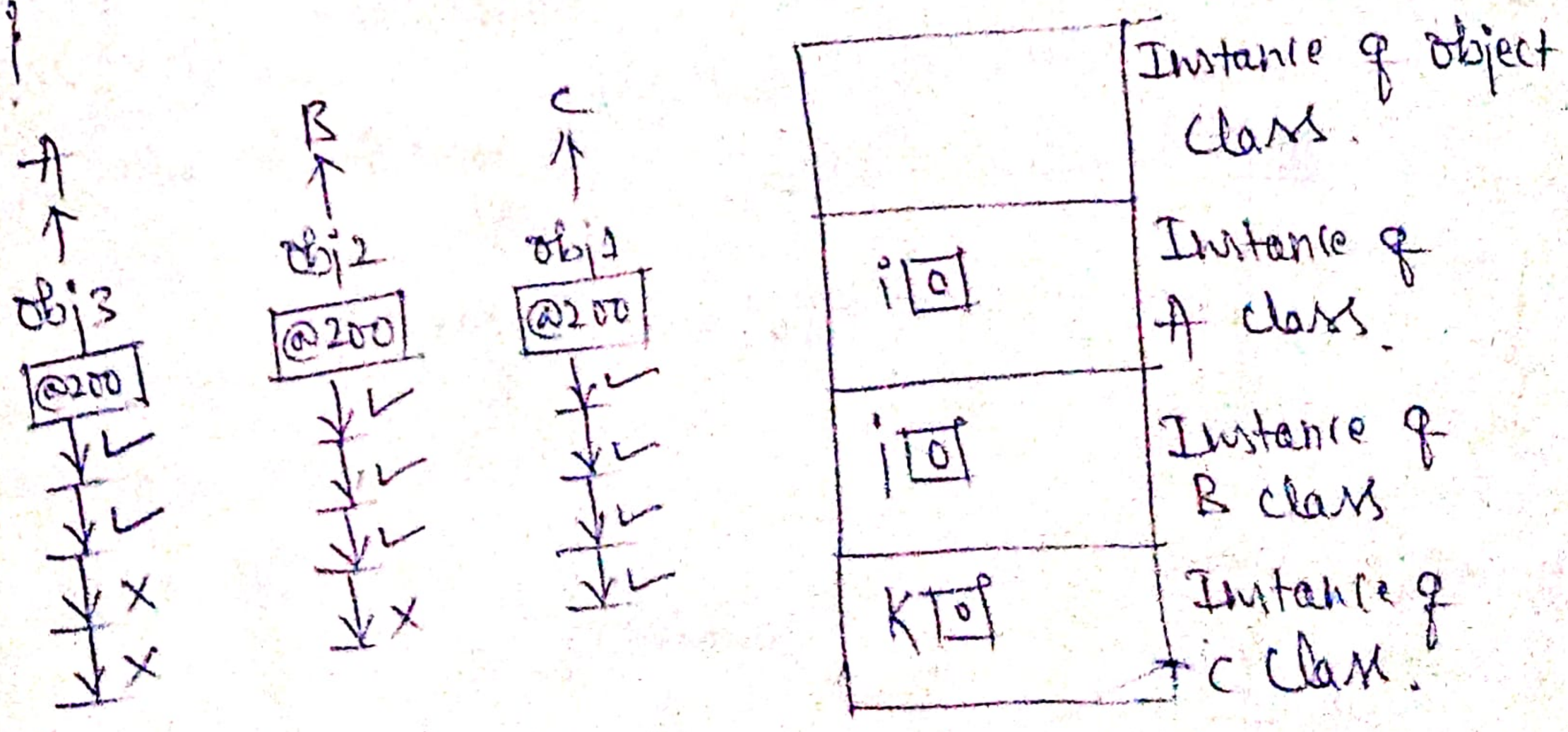
* Upcasting is automatically done by Compiler therefore it is also called as "Implicit type Casting".

```
class Driver1
{
    p.s.v.m()
    {
        B obj1 = new B();
        S.o.pln(obj1.i);//0
        S.o.pln(obj1.j);//0
        A obj2 = obj1;  //Up Casting.
        S.o.pln(obj2.i);//0
        S.o.pln(obj2.j);// CTE
    }
}
```



Note: Once after Upcasting Subclass type reference into Super class type using the upcasted reference we cannot access members of Sub class.

ex:
```
class Driver1
{
    p.s.v.m(String[] args)
    {
        C obj1 = new C();
        B obj2 = obj1; // upcasting
        A obj3 = obj1; // upcasting.

        S.o.println(obj1.i); // 0
        S.o.println(obj1.j); // 0
        S.o.println(obj1.k); // 0

        S.o.println(obj2.i); // 0
        S.o.println(obj2.j); // 0
        S.o.println(obj2.k); // CTE ⇒ obj is created for C then upcasting
                                ∴ we can't access C's variables.

        S.o.println(obj3.i); // 0
        S.o.println(obj3.j); // CTE
        S.o.println(obj3.k); // CTE
    }
}
```



A obj3 = obj1;
↓ ↓
A C
↓ ↓
Super class — Subclass.
upcasting.

B obj2 = obj1;
↓ ↓
B C
↓ ↓
Super class — Sub class
upcasting.

(ii) Down Casting

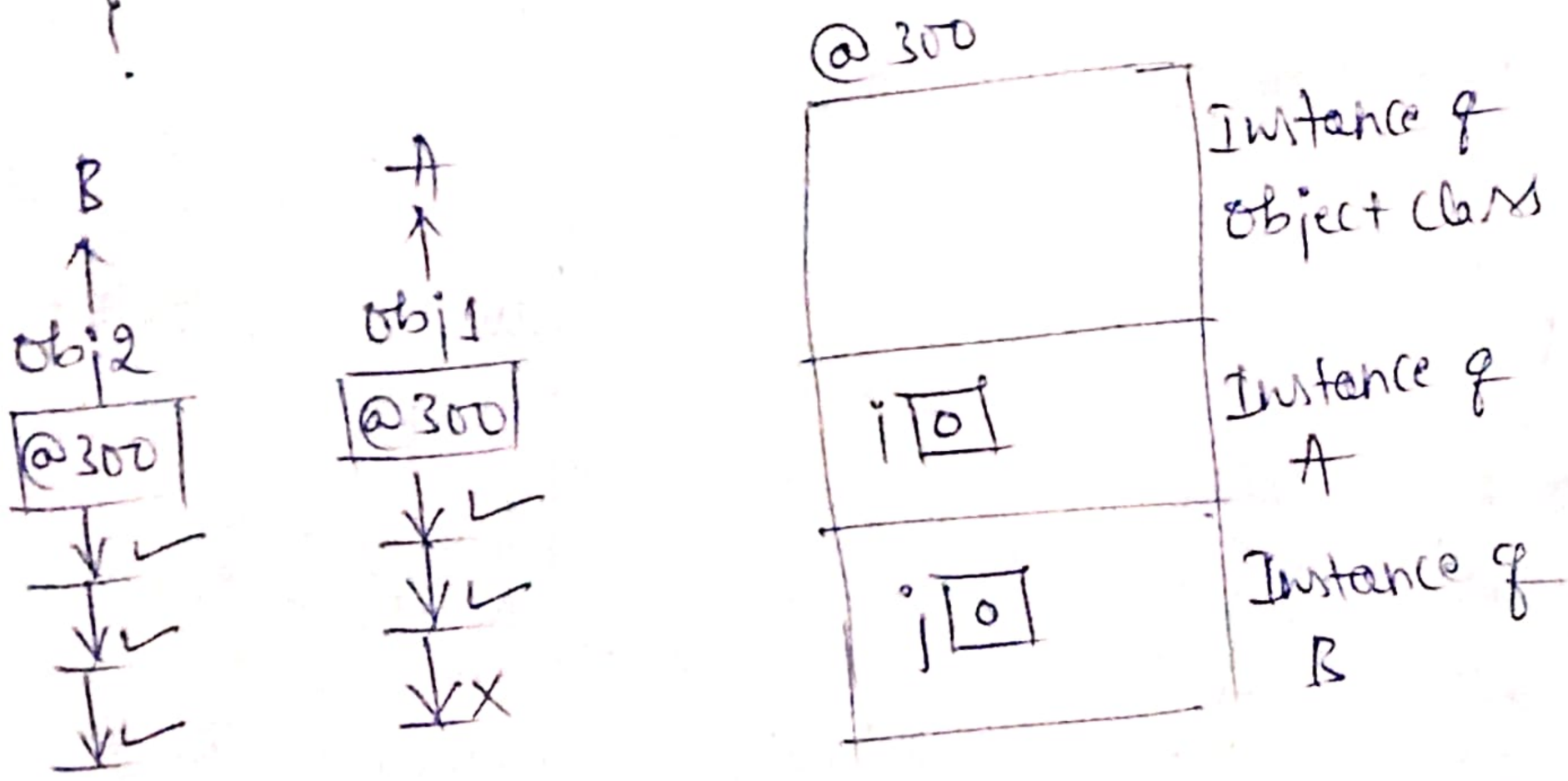
The process of Converting Super class type object reference to Sub Class type is Called down Casting.

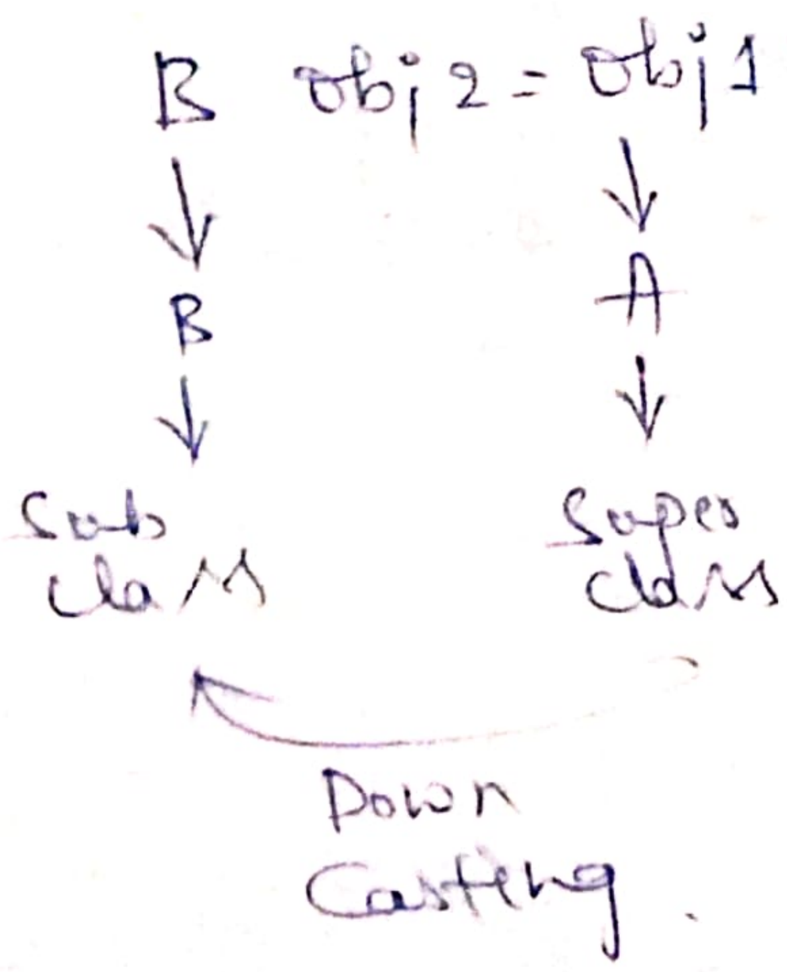
* Down Casting is not done by the Compiler it must be explicitly done with the help of Cast operator.

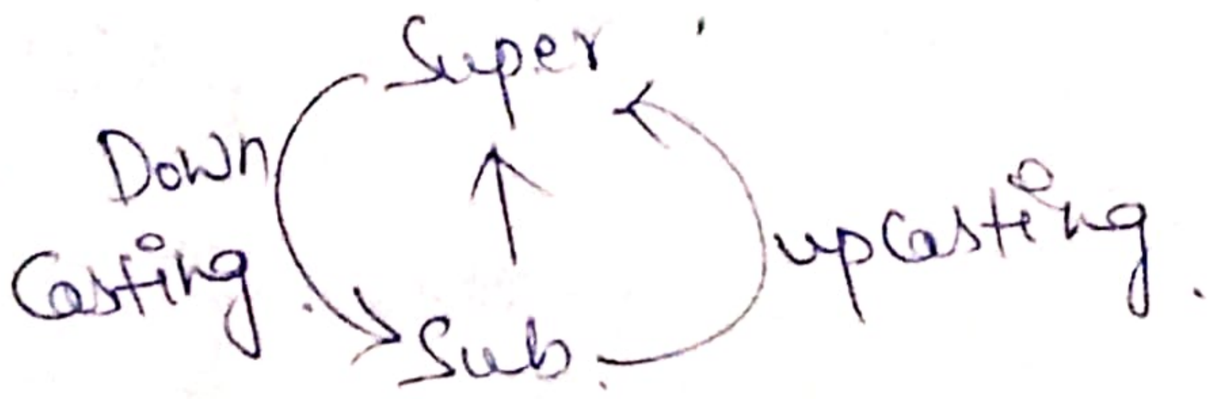
Syntax: (datatype) Variable;

ex:

```
class Driver1
{
    public Static Void main(String[] args)
    {
        A obj1 = new B();// upcasting
        S.o.pln(obj1.i);//0
        S.o.pln(obj1.j);// CTE
        B obj2 = (B)obj1;// down Casting
        S.o.pln(obj2.i);//0
        S.o.pln(obj2.j);//0
    }
}
```







```
class Driver1
{
    p.s.v.m(S[] args)
    {
        A obj1 = new C();  // up Casting
        B obj2 = (B) obj1;  // down Casting
        C obj3 = (C) obj2;  // down Casting

        s.o.pln(obj1.i);  //o
        s.o.pln(obj1.j);  //CTE
        s.o.pln(obj1.k);  //CTE

        s.o.pln(obj2.i);  //o
        s.o.pln(obj2.j);  //o
        s.o.pln(obj2.k);  //CTE

        s.o.pln(obj3.i);  //o
        s.o.pln(obj3.j);  //o
        s.o.pln(obj3.k);  //o
    }
}
```

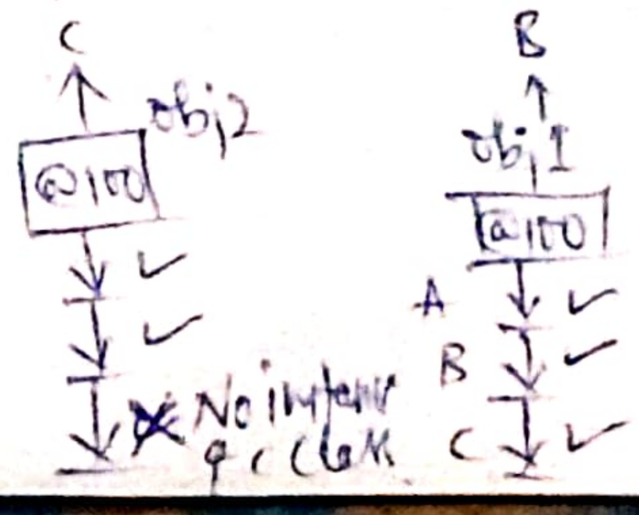
Note: So while downcasting there is a possibility of getting class cast exception.

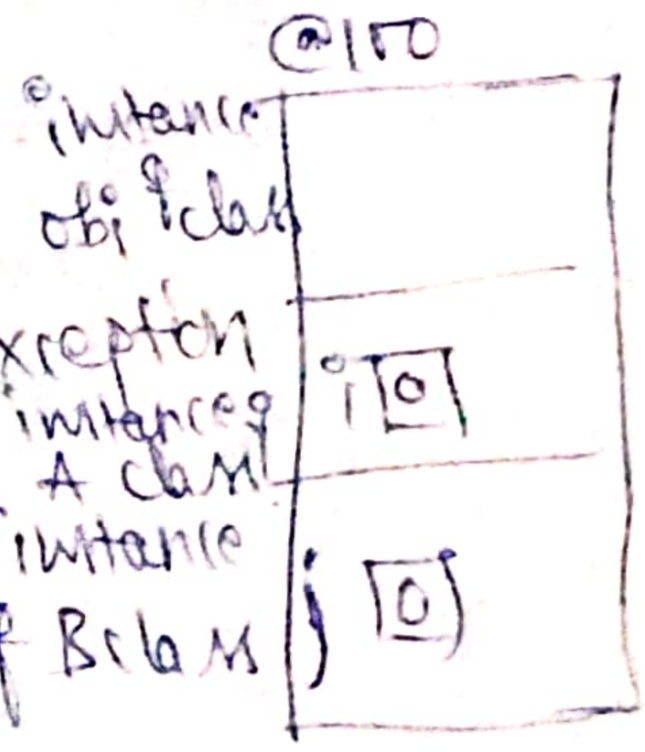
class cast exception

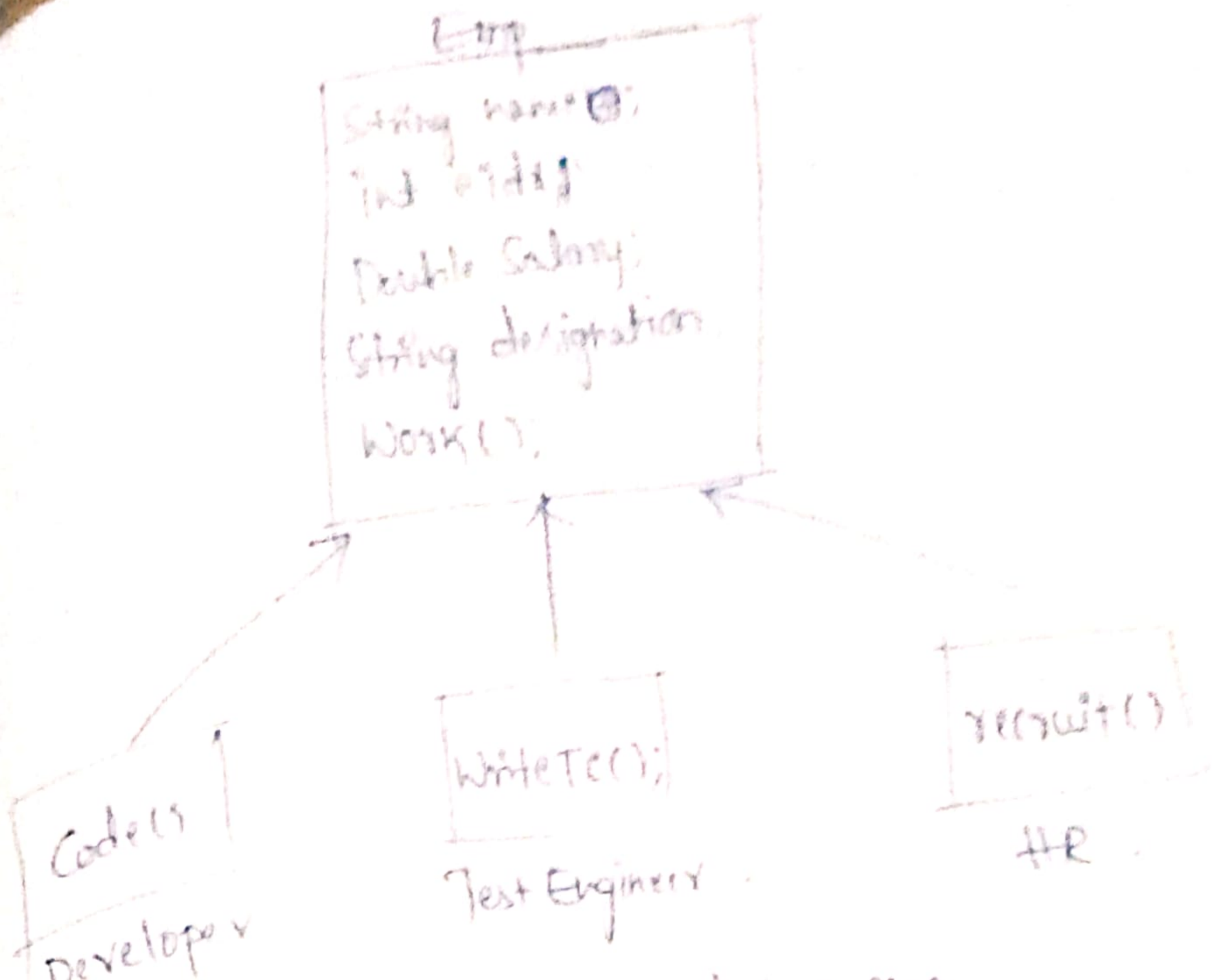
* when the object is created for super class if we try to downcast super class type object reference into sub class type since super class object will not have instance of subclass we will get class cast exception.

ex: class Driver1

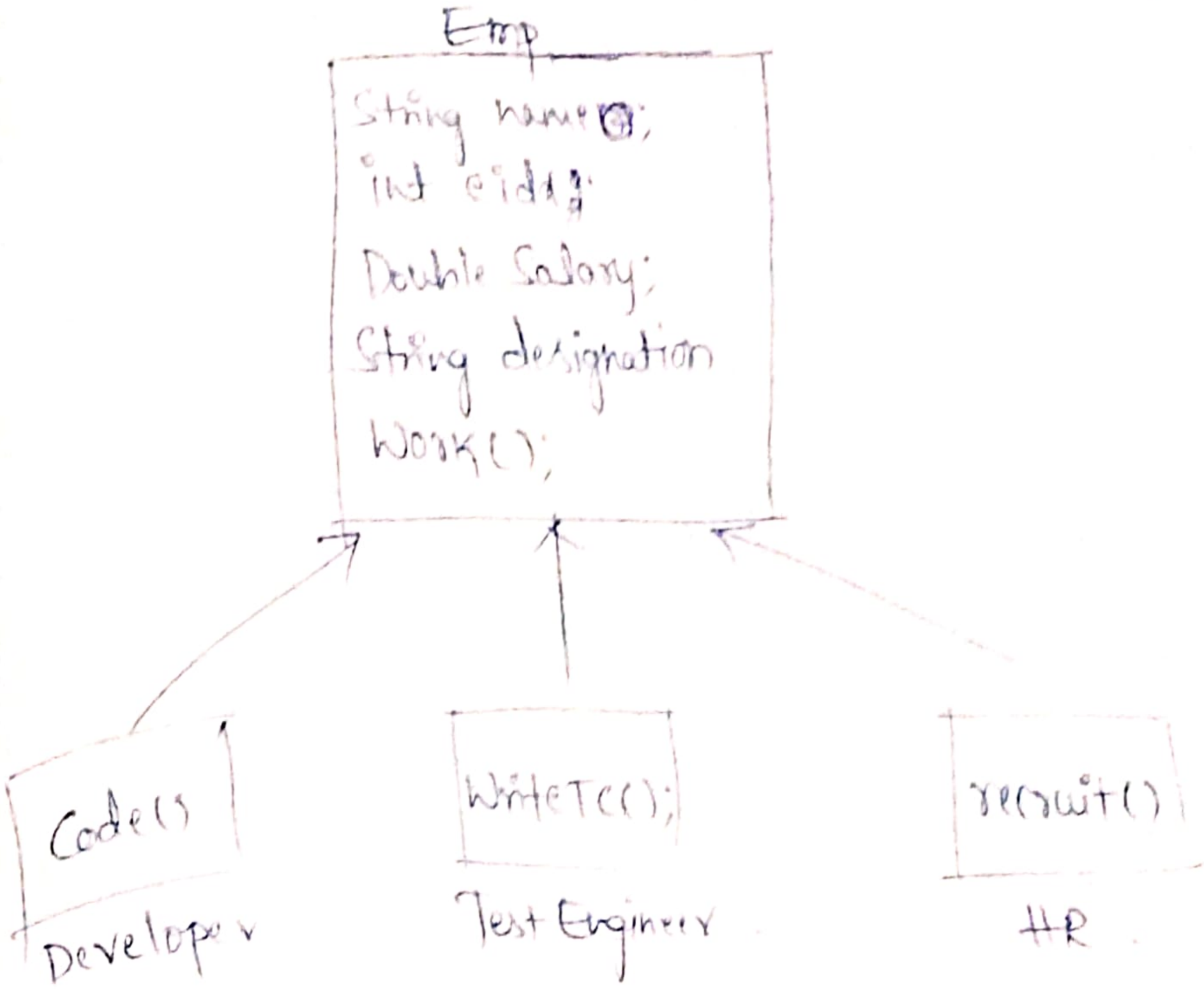
```
{
    p.s.v.m(SS[] args)
    {
        B obj1 = new B();
        C obj2 = (C) obj1;  // class Cast Exception
    }
}
```







* Achieve data hiding for all properties of emp.
* Eid & designation should be only readable
* Eid should be autogenerated.
* Salary should be modified only if newly entered Salary is greater than previous Salary.
* Design method which displays the details of the employee.
* Display total Count of employees.



* Achieve data hiding for all properties of emp.
* Eid & designation should be only readable
* Eid should be autogenerated.
* Salary should be modified only if newly entered Salary is greater than previous Salary.
* Design method which displays the details of the employee.
* Display total Count of employees.

```
class Driver1
{
    p.s.v.m()
    {
        A obj1 = new B(); //upcasting
        S.o.pln(obj1.i); //0
        S.o.pln(obj1.j); //CTE
        S.o.pln(((B)obj1).j); //0
        S.o.pln(((C)obj1).k); // Class Cast Exception
    }
}
```

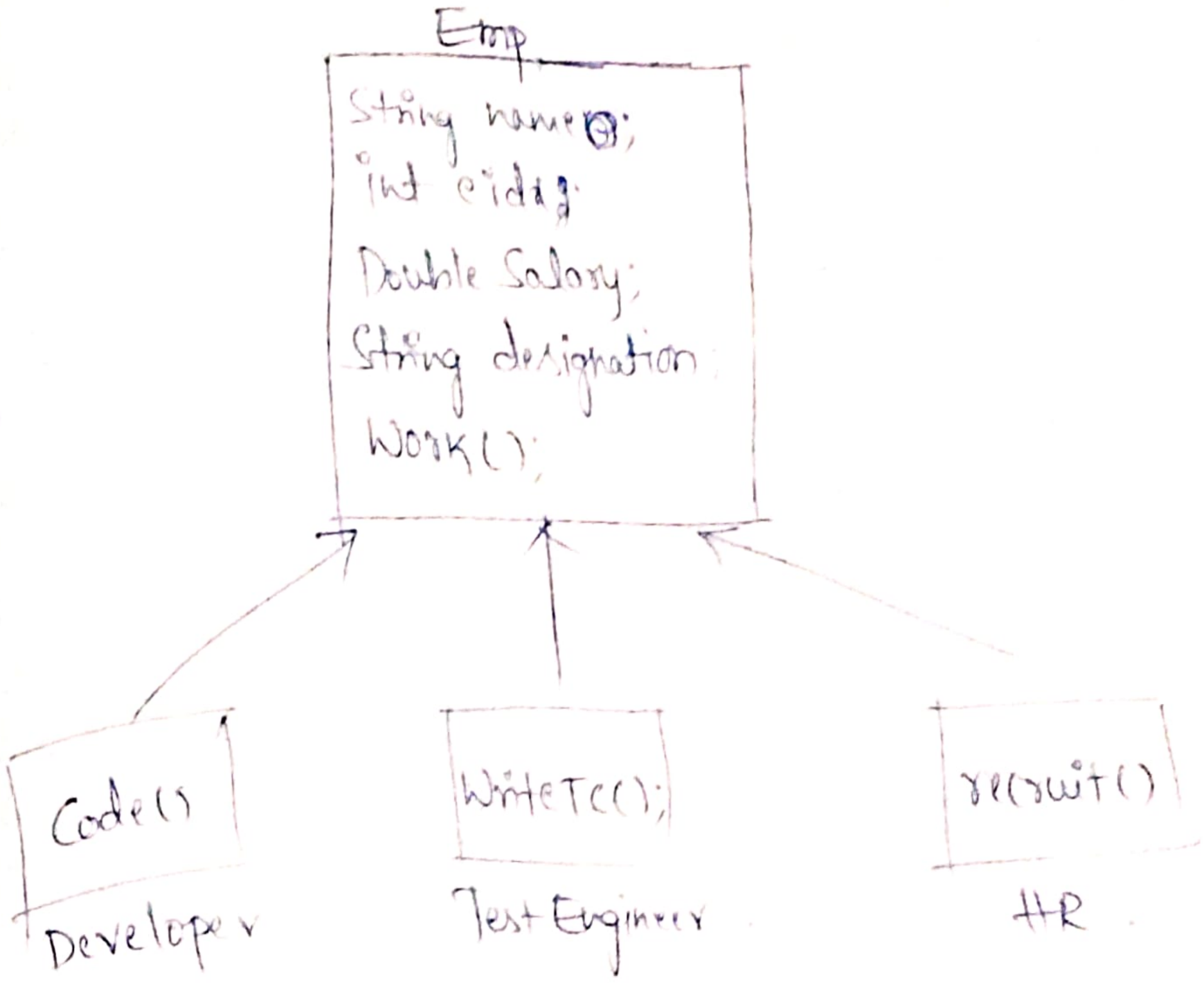
"Instanceof" Operator:-

* Instanceof Operator is used to check whether a given class instance is present in the object.

Syntax: object_reference instanceof classname

* Instanceof Operator returns true if the given class instance present in the object.
* It returns false if the instance of the given class is not present in the object.
* If the given class & the class for which object is created are not having is-a relationship then generates CTE.

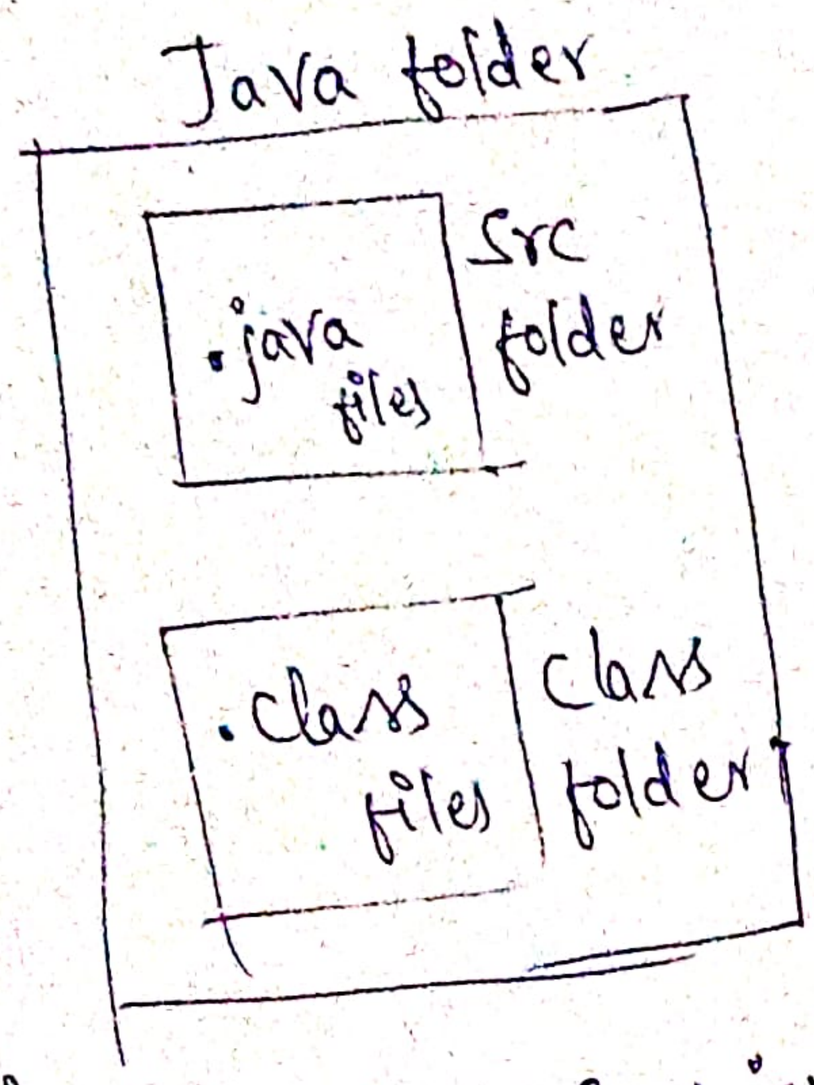
ex:
```
class Driver1
{
    p.s.v.m(S[] args)
    {
        B obj1 = new B();
        S.o.pln(obj1 instanceof object); //true
        S.o.pln(obj1 instanceof A); //true
        S.o.pln(obj1 instanceof B); //true
        S.o.pln(obj1 instanceof C); //false
        S.o.pln(obj1 instanceof Driver1); // CTE
    }
}
```

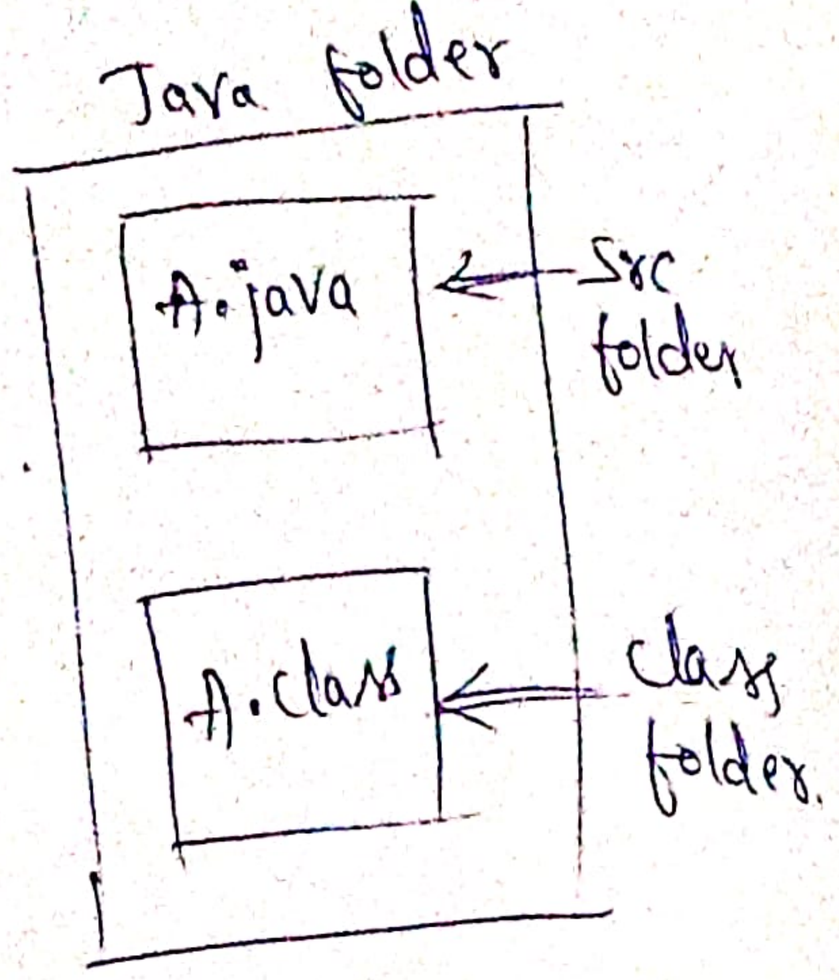


* Achieve data hiding for all properties of emp.
* Eid & designation should be only readable
* Eid should be autogenerated.
* Salary should be modified only if newly entered salary is greater than previous salary.
* Design method which displays the details of the employee.
* Display total Count of employees.

How to Separately Store .java file & .class file

* The folder which stores .java files is generally named as "Src"
* The folder which stores .class files is generally named as "Class"





ex
```
class A
{
    p.s.v.m()
    {
        s.o.pln("Hello");
    }
}
```

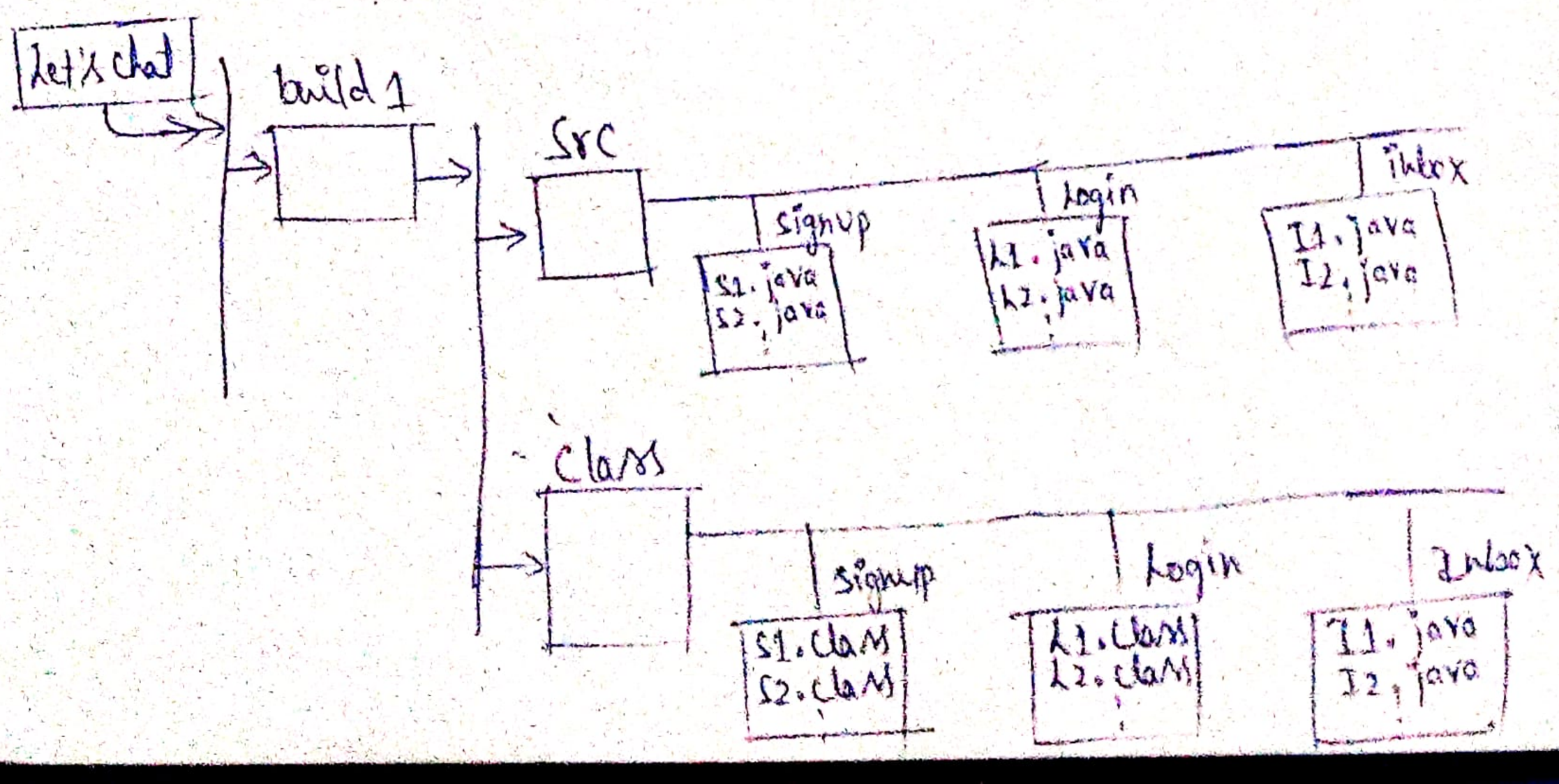
Src> javac -d ../class A.java ↵ Compile

Src> java -cp ../class A ↵ execute

Packages:

* Package is a collection of related .java files.
* With the help of packages we can organize the .java files.
* With the help of packages searching & accessing .java file becomes easy.
* Package can be created with the help of package keyword.

Syntax: package Package_name;



Ex

```
package SignUp;
Class S1
{
p.s.v.m( )
{s.o.pln("Signup Successfully!!");
}
}
```

S1.java
(Save in Src/signup)

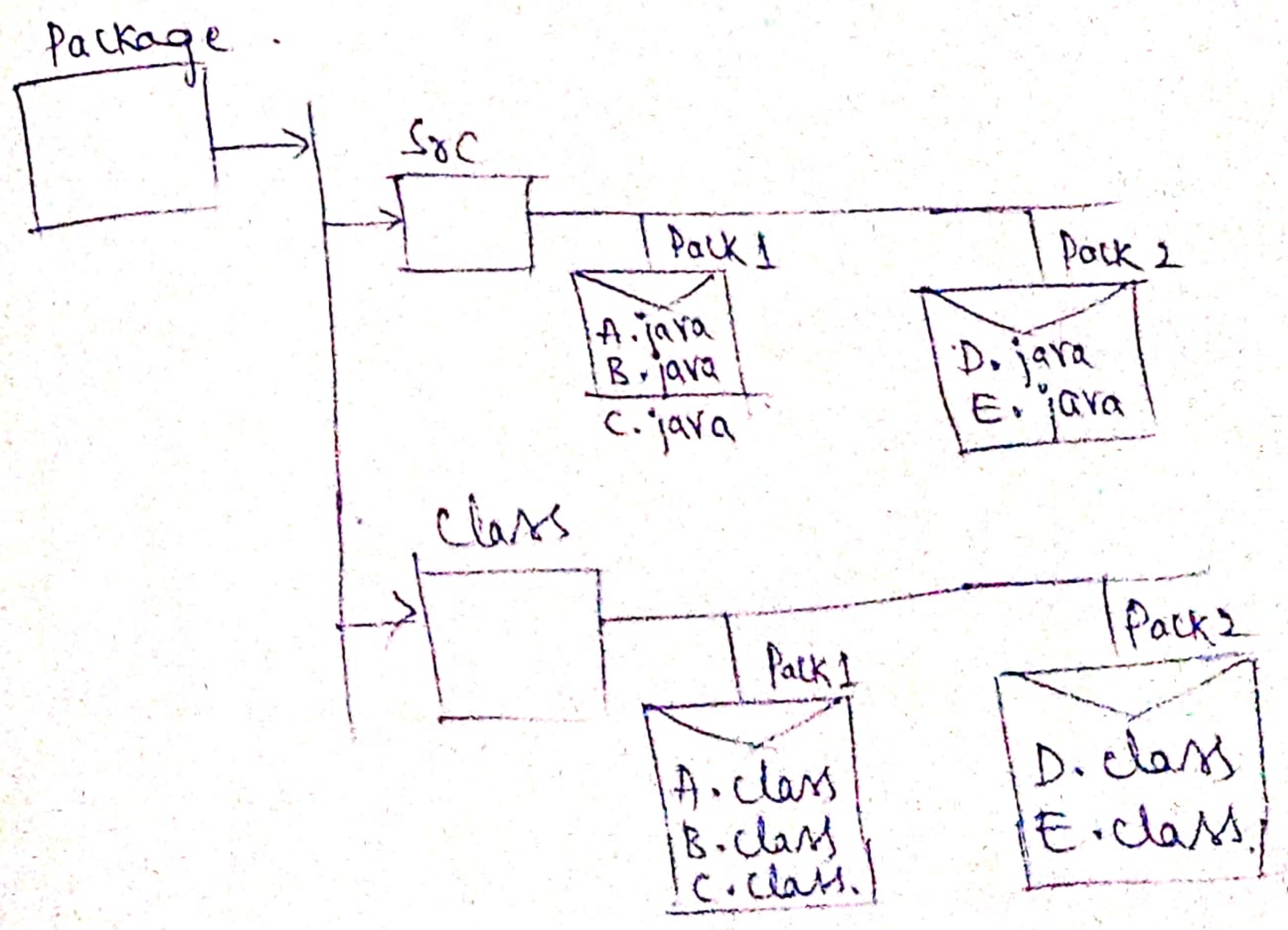
It will Create ~~Signup~~ S1.class file in SignUp folder ~~also~~ in class.

* In above example SignUp package Under Src must be created by programmer & S1.java file must be stored inside it.
* The SignUp package Under Class must be Created by Compiler & S1.Class must be stored inside it. this Can be achieved With the help of package Keyword.

Src> javac -d ../Class Signup/S1.java ↵ Compiling.
Src> java -cp ../class Signup.S1 ↵ (first it will goes to Class then Creates signup folder then S1.class will be Crea)
~~Sre~~ Signup Successfull!!!

* Package Keyword must be used as first statement of a java program.

Ex.



Access Specifiers/Modifiers :-

* are used to set the accessibility for the members of java pgm

* Private
* Default
* Protected
* Public

	Private	Default	Protected	Public
Inside the Same class	✓	✓	✓	✓
Inside the sub class of Same class	✗	✓	✓	✓
Inside the diff class of Same package	✗	✓	✓	✓
Inside the sub class of diff package	✗	✗	✓	✓
Inside the diff class of diff package	✗	✗	✗	✓

Method Overriding :

is a process of overriding the implementation of super class non static method by sub class non-static method implementation during object creation of sub class.

Rules for Overriding a method

1. Name of the super class method & sub class method must be same
2. Formal argument of superclass method & sub class method must be same.
3. Return type of super class method & sub class method must be same.
4. Access specifier of super class method & sub class method can be same (or) sub class method's access specifier must have higher visibility compare to super class method access specifier

How to use ~~package~~ class from one package to another

* We can use class from one package in another package in 2 ways.

(i) Using "import" keyword.

(a) * Syntax: import Package_name class_name;

(this imports only the specified class from given packa

(OR)

import Package_name.*;

(This imports all the classes from the given package)

(ii) Using "fully qualified name".

Syntax: packagename.classname.membername

Note: A class to be used outside the package must be a publ class.

Access Specifiers / Modifiers :-

* are used to set the accessibility for the members of java pgm

* Private
* Default
* Protected
* Public

	Private	Default	Protected	Public
Inside the same class	✓	✓	✓	✓
Inside the sub class of same class	✗	✓	✓	✓
Inside the diff class of same package	✗	✓	✓	✓
Inside the sub class of diff package	✗	✗	✓	✓
Inside the diff class of diff package	✗	✗	✗	✓

Method Overriding :

is a process of overriding the implementation of super class non-static method by sub class non-static method implementation during object creation of sub class.

Rules for overriding a method

1) Name of the super class method & sub class method must be same.
2) Formal argument of superclass method & subclass method must be same.
3) Return type of super class method & sub class method must be same.
4) Access specifier of super class method & sub class method can be same (or) sub class method's access specifier must have higher visibility compare to super class method access specifier

ex:
```
class A
{
    public Void test()
    {
        S.o.pln("From non-Static test() of A");
    }
}
class B extends A
{
    public Void test()
    {
        S.o.pln("From non-Static test() of B");
    }
}
class Driver1
{
    p.s.v. m()
    {
        A obj1 = new A();
        obj1.test(); // from non Static test() of A

        B obj2 = new B();
        obj2.test(); // from non-Static test() of B

        A obj3 = obj2; // upCasting
        obj3.test(); // from non-Static test() of B
    }
}
```

ex:
```
class Animal
{
    public Void talk()
    {
        S.o.pln("Animals Can talk");
    }
}
class Dog extends Animal
{
    public Void talk()
    {
        S.o.pln("Dog barks");
    }
}
```

```
class Lion extends Animals
{   public Void talk()
    {
      s.o.pln("Lion Roars");
    }

}
class Driver 2
{   p.s.v.m()
    {
      Animal a1 = new Animal();
          a1.talk(); //Animal Can talk

      Dog  a2 = new Dog();
           a2.talk(); // Dog barks

      Animal a3 = new Dog();
           a3.talk(); // Dog barks

      Lion a4 = new Lion();
           a4.talk(); // Lion roars

      Animal a5 = a4;
          a5.talk(); // Lion roars.

      Dog a6 = (Dog)a5; // class CastException.
           a6.talk();
    }

}
```

<u>Note</u>: Overriding happens with only non-static methods

Polymorphism:-

Polymorphism is the process of performing single action in different ways.

→ A statement executed multiple times generating different results is also considered as polymorphism.

2 types

1. Compile Time Polymorphism.
2. Run Time Polymorphism.

1. Compile Time Polymorphism:

ex: Method Overloading.

```
class Facebook
{
  public void signin(String mailid)
  {
    :
  }
  public void signin(int phno)
  {
    :
  }
}
```

```
class Driver1
{
  p.s.v.m()
  {
    Facebook f = new facebook();
    f.signin(9874216593);
  }
}
```

* In Compile time polymorphism link b/w (or) bind b/w method call statement & method implementation happends during compilation, based on the arguments passed in the method call statement.

(2) Run Time Polymorphism :-

In run time polymorphism the link b/w (or) bind b/w method Call statement & method implementation happends during run time (execution).

We Can achieve run time polymorphism using

(i) Is-a-relationship

(ii) Upcasting.

(iii) Method Overriding.

Ex:

```
class Card
{
  public Void makePayment()
  {
    S.O.pln("Making Payment");
  }
}

class DebitCard extends Card
{
  public Void makepayment()
  {
    S.O.pln("making payment
            through DebitCard");
  }
}

class CreditCard extends Card
{
  public Void makepayment()
  {
    S.O.pln("making payment
            through CreditCard");
  }
}
```

```
class SwipingMachine
{
  public Void Swipe(Card c)
  {
    c.makePayment();
  }
}

class Driver1
{
  P.S.V.M()
  {
    DebitCard C1 = new DebitCard();
    SwipingMachine s = new SwipingMachine;
    S.swipe(C1); // making Payment
                    through debitCard.
    CreditCard C2 = new CreditCard();
    S.swipe(C2); // making Payment
                    through CreditCard
  }
}
```

Method Shadowing

* We can have same static methods in super class & sub class here which method implementation is used for method call statement depends on type of object reference variable is used. This process is known as method shadowing.

```
class A
{
  public static void test()
  {
    s.o.pln("from test() of A");
  }
}
class B extends A
{
  public static void test()
  {
    s.o.pln("from test() of B");
  }
}
```

```
class Driver2
{
  p.s.v.m()
  {
    A obj1 = new A();
    obj1.test(); //from A
    B obj2 = new B();
    obj2.test(); //from B
    A obj3 = obj2; //upcast
    obj3.test(); //from A
  }
}
```

Variable Shadowing:

* We can have same static variables in super() & subclass also we can have same non-static variables in super class & sub class here result depends on type of object reference. Variable we use this known as variable shadowing

```
class A
{
  static int a=10;
  int b=20;
}
class B extends A
{
  static int a=100;
  int b=200;
}
```

```
class Driver2
{
  p.s.v.m(String[] args)
  {
    B obj1 = new B();
    s.o.pln(obj1.a); //100
    s.o.pln(obj1.b); //200
    A obj2 = obj1;
    s.o.pln(obj2.a); //10
    s.o.pln(obj2.b); //20
  }
}
```

```
class super
{ static int i=1;
  int j=2;
  public static void test1()
  {
   s.o.pln("from test1() of Super class");
  }
  public void test2()
  {
   s.o.pln("from non-static test2()
                of super class");
  }
}
class Sub extends super.
{ static int i=10;
  int j=20;
  public static void test1():
  {
   ~~super~~ s.o.pln("from test1() of Sub class").
  }
  public void test2()
  {
   s.o.pln("from non-static test2()
                of Sub class");
  }
}
class Driver3
{
  p.s.v.m(SS3 a)
  {
   Super obj1 = new Sub();
   s.o.pln(obj1.i); // 1
   s.o.pln(obj1.j); // 2.
   obj1.test1(); // from static test1()
                    of Super class
   obj1.test2(); // from non-static test2()
                    of Sub class.
  }
}
```

Final :-

is a Keyword in java it acts as modifier & it is used to Create Constants.

Final With local Variables :

* A local Variable declared as final Cannot be re-initialize

```
class A
{
  p s.v.m(S[] args)
  {
    final int a=10;
    S.o.pln(a);//10
    a=200;//CTE
    S.o.pln(a);
  }
}
```

Note:- For final Variable We Can't use increment (or) decrement operator.

ex. final int a=10;
S.o.pln(a++);// CTE

Final With formal arguments :

We Can declare Variable present in method header as a final Variable but we Cannot re-initialize.

ex:-
```
class A
{
  p.s.v.m( )
  {
    test(10);
  }

  public static Void test(final int a)
  {
    S.o.pln(a);//10
    a=20;//CTE
    S.o.pln(a);
  }
}
```

Final With global Variables:

(a) Static Variables:

* We Can make a Static Variable final but We must initialize it using Static initialize.
* Once We declared a Static Variable as final We Cannot re-initialized.

ex:
```
class A
{
    final static int a;
    static
    {
        a=10;
    }
    public s.v.m()
    {
        s.o.pln(a);
        a=20; //CTE
    }
}
```

ex:
```
class A
{
    final static int a; //CTE
    p.s.v.m(S[] a)
    {
        s.o.pln(a);
    }
}
```

(b) Non-Static Variables:

* We Can make non-Static Variable final but it must be initialize using non-Static initialize. otherwise Compiler generates CTE.
* Once We make a non-Static Variable final it Can't be re-initialize

Final With Methods

Non-Static methods:

* We Can make non-Static method final for final non-Static method Can Override it's Super class method.
* A final non-Static method Cannot be Override by Sub-class non Static method.

```
class A
{
    public void test()
    {
        S.o.pln("A");
    }
}
class B extends A
{
    final public void test()
    {
        S.o.pln("B");
    }
}
```

```
class Driver
{
    P.S.V.m(S[] args)
    {
        B Obj1 = new B();
        Obj1.test();//B
    }
}
```

ex:

```
class A
{
    final public void test()
    {
        S.o.pln("A");
    }
}
class B extends A
{
    public void test()//CTE
    {
        S.o.pln("B");
    }
}
```

Final With Class:

We can declare class as final but it cannot be initialized

We cannot create sub class for final class.

ex:

```
final class C
{
    :
}
```

```
class D extends C //CTE
{
    :
}
```

Note: final class can extend another class.

Final With Static methods:

* We can take static method final but we cannot achieve method shadowing.

Final	Finally	Finalize
* It is a keyword, used to apply restriction on class, method, or Variable.	* It is a block used to execute imp code whether exception occurs or not (Exception Handling)	* It is a method used to perform clean up processing, before object is garbage collected
* Used with classes, methods & Variables	* It is always related to try & catch block in exception handling.	* It is used with the objects.
* Once it's declared final variable becomes Constant & Cannot be modified.	* It runs imp. code even if exception occurs or not	* It performs cleaning activities with respect to the object before it's destruction.
* Final method is executed only when we call it	* Finally block is executed as soon as the try-catch block is executed.	* Finalize method executed just before the object is destroyed.